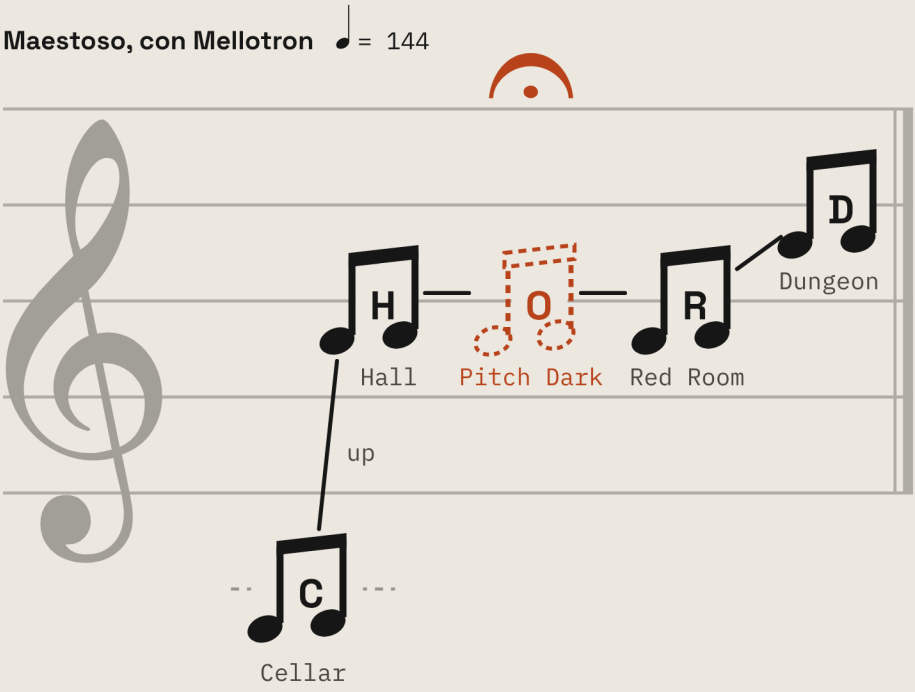
chord

AN INTERACTIVE FICTION MODELING LANGUAGE





Writing Interactive Fiction with Chord

David Cornelson

First Edition (for Chord 4.1.0)

The Chord Manual

Writing Interactive Fiction in Chord

Chord 4.1.0, Sharpee 5.7.0

David Cornelson

The Chord Manual

Writing Interactive Fiction in Chord

Published by David Cornelson

Copyright © 2026 David Cornelson. All rights reserved. No part of this book may be reproduced in any form without written permission from the publisher, except for brief quotations used in reviews.

This edition is pinned to Chord 4.1.0, as shipped in Sharpee 5.7.0. Every example in it is compiled and played against that release.

- Preface
    - 0.1 What this book promises
    - 0.2 The edition
    - 0.3 Two kinds of reader
- How to read this book
    - 0.4 Where to start
    - 0.5 The eight parts
    - 0.6 The two kinds of box
    - 0.7 Conventions
    - 0.8 Build and try it
    - 0.9 The companion folders
    - 0.10 When something is missing
- Part I. A first story
- 1 The shape of a story file
    - 1.1 One line, one thing
    - 1.2 Indentation says what belongs to what
    - 1.3 How a block ends
    - 1.4 Blank lines
    - 1.5 Comments
    - 1.6 Words, numbers and quotation marks
    - 1.7 What you have so far
- 2 The story header
    - 2.1 The two fields every header needs
    - 2.2 What the player sees
    - 2.3 Version, description and credits
    - 2.4 The prologue
    - 2.5 The two identifiers
    - 2.6 When a field is refused
    - 2.7 What you have so far
- 3 The player and the start
    - 3.1 Making Ansel
    - 3.2 Handing over the role
    - 3.3 What the start block is not for
    - 3.4 The opening

    - 3.5 When it goes wrong
    - 3.6 What you have so far
- 4 Build, play, read an error
    - 4.1 Installing the command
    - 4.2 A folder for the story
    - 4.3 Checking the story
    - 4.4 Playing the story
    - 4.5 Reading an error
    - 4.6 When the compiler guesses
    - 4.7 Warnings
    - 4.8 Two mistakes with the folder
    - 4.9 What you have so far
- Part II. The world
- 5 Rooms and exits
    - 5.1 What `create` does
    - 5.2 Exits
    - 5.3 The way back is free
    - 5.4 One-way exits
    - 5.5 The rest of Lanternhall
    - 5.6 The first visit
    - 5.7 A heading of its own
    - 5.8 Build and try it
    - 5.9 What you have so far
- 6 Things
    - 6.1 A thing and where it is
    - 6.2 Things that stay put
    - 6.3 More than one, and things with names
    - 6.4 Other words for a thing
    - 6.5 A thing in several rooms
    - 6.6 What Ansel carries
    - 6.7 Hidden things
    - 6.8 Build and try it
    - 6.9 What you have so far
- 7 Containers and supporters
    - 7.1 Two kinds that hold

    - 7.2 Opening and closing
    - 7.3 How much it holds
    - 7.4 A cupboard on the wall
    - 7.5 Seeing through, and saying it yourself
    - 7.6 Getting into things
    - 7.7 Build and try it
    - 7.8 What you have so far
- 8 Doors, locks and keys
    - 8.1 A door
    - 8.2 A lock and its key
    - 8.3 Playing it
    - 8.4 Saving the player a step
    - 8.5 Locks on other things
    - 8.6 Naming keys
    - 8.7 Build and try it
    - 8.8 What you have so far
- 9 What things can do: the trait words
    - 9.1 Wearing
    - 9.2 Reading
    - 9.3 Eating and drinking
    - 9.4 Pulling and pushing
    - 9.5 Breaking
    - 9.6 Fixed to something
    - 9.7 Switching on and off
    - 9.8 The ones that wait
    - 9.9 Build and try it
    - 9.10 What you have so far
- 10 Light, dark and the other senses
    - 10.1 A dark room
    - 10.2 A light to carry
    - 10.3 What this does to the story
    - 10.4 Seeing in the dark
    - 10.5 Smell and sound
    - 10.6 Build and try it
    - 10.7 What you have so far

- 11 Regions
    - 11.1 Making a region
    - 11.2 Putting rooms in it
    - 11.3 Scenery for a whole region
    - 11.4 A run of rooms
    - 11.5 A table of exits
    - 11.6 A backdrop for the stacks
    - 11.7 Build and try it
    - 11.8 What you have so far
- 12 Ways that are closed, hidden or strange
    - 12.1 A way that refuses
    - 12.2 A way that kills
    - 12.3 A way that is sometimes there
    - 12.4 A way with its own command
    - 12.5 Build and try it
    - 12.6 What you have so far
- Part III. Making things happen
- 13 States
    - 13.1 Declaring states
    - 13.2 Naming a state
    - 13.3 Reading a state
    - 13.4 Changing a state
    - 13.5 States that go both ways
    - 13.6 The story's own states
    - 13.7 The states you do not declare
    - 13.8 Build and try it
    - 13.9 What you have so far
- 14 Reacting to the player: on and after
    - 14.1 A thing that says no
    - 14.2 Where a clause lives
    - 14.3 Speaking in the action's place
    - 14.4 Adding to an action: after
    - 14.5 Several actions, one clause
    - 14.6 The other thing in the command
    - 14.7 Conditions and once

    - 14.8 Clauses about the person
    - 14.9 Build and try it
    - 14.10 What you have so far
- 15 What a clause can say and do
    - 15.1 Requirements: must
    - 15.2 Prohibitions: refuse when
    - 15.3 Printing: phrase
    - 15.4 Changing and moving
    - 15.5 The when suffix
    - 15.6 It, and its absence
    - 15.7 Build and try it
    - 15.8 What you have so far
- 16 Conditions
    - 16.1 What a condition can ask
    - 16.2 Joining conditions
    - 16.3 Naming a condition
    - 16.4 Conditions with a gap: it
    - 16.5 Any and no
    - 16.6 Each
    - 16.7 What a condition cannot ask
    - 16.8 Build and try it
    - 16.9 What you have so far
- 17 Choosing between outcomes
    - 17.1 Choosing by state: select on
    - 17.2 Refusals stay outside
    - 17.3 One clause, several answers
    - 17.4 Choosing by turn: select with a strategy
    - 17.5 Choosing by count: first time, second time
    - 17.6 Which one to use
    - 17.7 Build and try it
    - 17.8 What you have so far
- 18 Counting, scoring and ending
    - 18.1 A counter on a thing
    - 18.2 A counter for the story
    - 18.3 Scoring

    - 18.4 Ranks
    - 18.5 Winning and losing
    - 18.6 Trying the endings
    - 18.7 Killing the player
    - 18.8 Build and try it
    - 18.9 What you have so far
- 19 Time
    - 19.1 Every turn
    - 19.2 Timers
    - 19.3 More about timers
    - 19.4 A sequence
    - 19.5 Chapters
    - 19.6 When something moves
    - 19.7 Build and try it
    - 19.8 What you have so far
- Part IV. Words
- 20 Phrases
    - 20.1 Two ways to declare a phrase
    - 20.2 Variants and strategies
    - 20.3 The sayings of Hobb
    - 20.4 Text that keeps its shape
    - 20.5 Markers
    - 20.6 A phrase inside a description
    - 20.7 Build and try it
    - 20.8 What you have so far
- 21 A thing's own words
    - 21.1 Detail: a sentence that depends
    - 21.2 Detail or a clause?
    - 21.3 Present: how a thing is announced
    - 21.4 Disappeared: a parting line
    - 21.5 The general rule: a thing may answer for itself
    - 21.6 Build and try it
    - 21.7 What you have so far
- 22 Changing the standard messages
    - 22.1 Finding a message

    - 22.2 Overriding one message
    - 22.3 An override is a phrase
    - 22.4 Several at once
    - 22.5 The markers in a template
    - 22.6 How much to override
    - 22.7 Phrasebooks
    - 22.8 When to reach for a phrasebook
    - 22.9 Phrases in a file of their own
    - 22.10 Build and try it
    - 22.11 What you have so far
- 23 New verbs
    - 23.1 Declaring an action
    - 23.2 The grammar
    - 23.3 What must be true of the thing named
    - 23.4 What must be true of the player
    - 23.5 What an action may not ask
    - 23.6 When nothing answers
    - 23.7 The four workings
    - 23.8 A verb that only answers
    - 23.9 Verbs with a subject
    - 23.10 Other kinds of slot
    - 23.11 A verb that comes and goes
    - 23.12 Why a thing cannot simply answer
    - 23.13 Build and try it
    - 23.14 What you have so far
- 24 New kinds of thing
    - 24.1 A trait that answers a verb
    - 24.2 One trait, several things
    - 24.3 A trait may answer standard actions too
    - 24.4 A trait with one thing
    - 24.5 A trait with states and a score
    - 24.6 A thing's own words, again
    - 24.7 The bell and the grille
    - 24.8 Data on a trait
    - 24.9 Is it a wick?

    - 24.10 Build and try it
    - 24.11 What you have so far
- 25 Bending the standard grammar
    - 25.1 Adding words to an action
    - 25.2 Extending a standard action
    - 25.3 Taking words away
    - 25.4 Moving a word
    - 25.5 What to add
    - 25.6 A file for the grammar
    - 25.7 Build and try it
    - 25.8 What you have so far
- Part V. People
- 26 People who act
    - 26.1 Pronouns
    - 26.2 The stock behaviours
    - 26.3 Having a person do something
    - 26.4 Goals
    - 26.5 Build and try it
    - 26.6 What you have so far
- 27 Asking and telling
    - 27.1 A table of topics
    - 27.2 Looking things up
    - 27.3 Greetings
    - 27.4 Manner
    - 27.5 Build and try it
    - 27.6 What you have so far
- 28 Conversations with a shape
    - 28.1 A thread
    - 28.2 A question
    - 28.3 The conclusion
    - 28.4 A way back
    - 28.5 What a table can remember
    - 28.6 Speaking first
    - 28.7 Build and try it
    - 28.8 What you have so far

- 29 Inner lives
    - 29.1 Mood
    - 29.2 Feelings
    - 29.3 What a person knows
    - 29.4 Build and try it
    - 29.5 What you have so far
- 30 Principles
    - 30.1 Something told in confidence
    - 30.2 A thing a person will not do
    - 30.3 What decides between principles
    - 30.4 Honesty
    - 30.5 What the lines add up to
    - 30.6 Build and try it
    - 30.7 What you have so far
- 31 More than one protagonist
    - 31.1 Two people who can be played
    - 31.2 Handing the role over
    - 31.3 The player is whoever has the role
    - 31.4 Handing it back
    - 31.5 When to do this
    - 31.6 Build and try it
    - 31.7 What you have so far
- Part VI. Larger machinery
- 32 Machines
    - 32.1 The parts
    - 32.2 The machine
    - 32.3 Moving between states
    - 32.4 Entering and leaving
    - 32.5 Telling the world
    - 32.6 Build and try it
    - 32.7 What you have so far
- 33 Vehicles
    - 33.1 A problem for a vehicle
    - 33.2 The book hoist
    - 33.3 A vehicle that is driven

    - 33.4 A vehicle the player steers
    - 33.5 A vehicle with rooms inside
    - 33.6 Build and try it
    - 33.7 What you have so far
- 34 Open water and open ground
    - 34.1 Turning the system on
    - 34.2 An expanse
    - 34.3 A vehicle on open water
    - 34.4 Looking across water
    - 34.5 Wind
    - 34.6 Things adrift
    - 34.7 The new ending
    - 34.8 Build and try it
    - 34.9 What you have so far
- 35 The opt-in systems
    - 35.1 What `use` means
    - 35.2 Combat
    - 35.3 Hunger
    - 35.4 Choosing
    - 35.5 Build and try it
    - 35.6 What you have so far
- Part VII. Presentation
- 36 Sound, music and pictures
    - 36.1 Declaring an asset
    - 36.2 Playing a sound
    - 36.3 A sound is also a sound in the world
    - 36.4 What a sound sounds like
    - 36.5 From afar, and with no file
    - 36.6 A bed of sound
    - 36.7 Showing a picture
    - 36.8 Music, named beds and layers
    - 36.9 Asking what the client can do
    - 36.10 Build and try it
    - 36.11 What you have so far

- 37 The page the player sees
    - 37.1 The client fields
    - 37.2 Building the page
    - 37.3 A channel
    - 37.4 Raising an event
    - 37.5 What a channel may return
    - 37.6 A reference in a field
    - 37.7 The stylesheet
    - 37.8 A page of your own
    - 37.9 Build and try it
    - 37.10 What you have so far
- Part VIII. Proving and shipping
- 38 Test trees
    - 38.1 Two ways to make one
    - 38.2 The folder
    - 38.3 A segment and its cards
    - 38.4 Reading what the story says
    - 38.5 What a card can assert
    - 38.6 Running the tree
    - 38.7 A fork
    - 38.8 An ending
    - 38.9 The whole tree, and what it covers
    - 38.10 Build and try it
    - 38.11 What you have so far
- 39 Claims
    - 39.1 A fact worth keeping
    - 39.2 Locking it
    - 39.3 The claims file
    - 39.4 What a claim may state
    - 39.5 The report
    - 39.6 When a lock breaks
    - 39.7 A claim written by hand
    - 39.8 A claim no line reaches
    - 39.9 The needs lines
    - 39.10 Which claims to keep

    - 39.11 Build and try it
    - 39.12 What you have so far
- 40 Publishing
    - 40.1 The identifier
    - 40.2 The version
    - 40.3 Whether the source goes too
    - 40.4 Publish
    - 40.5 What ships
    - 40.6 After the first release
    - 40.7 Build and try it
    - 40.8 What you have
- Appendix A. The grammar
- Appendix B. Kinds, trait words, state words and their pairings
- Appendix C. The standard actions and their gerunds
- Appendix D. Statements and conditions
- Appendix E. Character vocabulary
- Appendix F. Message aliases
- Appendix G. Removed forms and their replacements

Preface

Chord is a language for writing parser-based interactive fiction. A Chord story is a plain text file. It says what rooms there are and what is in them, who the people are and what they will talk about, what happens when the player does this or that, and how the story ends. The Sharpee platform reads that file and turns it into a game that plays in a terminal and in a web browser.

Chord was made for writers. Its lines read as sentences about a world, and none of them is program code in the usual sense. Nothing in it asks you to know how the platform underneath is built, and nothing in this book does either.

This book teaches Chord by building one story, **The Relighting**, from an empty file to a finished game with sound, a picture, a page of its own, a suite of tests and a release. The story is set in a school of magic that has stood shut for thirty years, on the morning a former student comes back to it. It grows by one idea a chapter, and at the end of every chapter it is a complete story that compiles and plays.

0.1 What this book promises

Every example in these pages was compiled, and every line of story output was copied from a real play of the story at the end of its chapter. A companion folder holds the story as it stands at the end of each chapter, with a test that replays the chapter's commands and checks what the story said.

Where a part of the language could not be shown working, the chapter leaves it out. A manual that describes what a language is supposed to do is easy to write and teaches people things that are not so. This one describes what the language did when it was asked.

0.2 The edition

This edition is pinned to Chord 4.1.0, as shipped in Sharpee 5.7.0. The language will go on changing, and a later edition will follow it. If your copy of the tools reports a later version, most of this book will still hold, and the release notes will tell you what does not.

0.3 Two kinds of reader

Some readers of this book have never written interactive fiction. The main text is written for them, and it assumes nothing but a text editor and patience.

Other readers have written a good deal of it, in Inform or in TADS, and want to know how Chord differs from what they know. There are short boxed notes for them throughout. The next page explains how the two kinds of note are marked, and which you can skip.

How to read this book

0.4 Where to start

If you are	Read
New to interactive fiction, or new to writing it	The chapters in order, skipping every boxed note.
An author who knows Inform or TADS	The chapters in order, with the boxes headed “From another system”.
Curious about what the platform does with your lines	The chapters in order, with the boxes headed “Under the Hood”.
Looking something up	The table of contents, and the appendices at the back.

The chapters build on one another, and the story grows through them, so the book works best read from the front. Part I is short and gets a story on the screen. If you read nothing else before starting a story of your own, read Parts I and II.

0.5 The eight parts

- **Part I. A first story** (chapters 1 to 4) covers the story file, its header, the player, and how to build a story, play it, and read an error.
- **Part II. The world** (chapters 5 to 12) covers rooms, things, containers, doors, light and darkness, regions, and ways that are closed or hidden.
- **Part III. Making things happen** (chapters 13 to 19) covers states, the clauses that react to the player, conditions, choices, scoring, endings and time.

- **Part IV. Words** (chapters 20 to 25) covers the story's prose, the platform's standard messages, and new verbs and new kinds of thing.
- **Part V. People** (chapters 26 to 31) covers people who act, talk, remember and feel, and stories with more than one protagonist.
- **Part VI. Larger machinery** (chapters 32 to 35) covers machines, vehicles, open water, and the systems a story may opt into.
- **Part VII. Presentation** (chapters 36 and 37) covers sound, pictures and the page in the browser.
- **Part VIII. Proving and shipping** (chapters 38 to 40) covers tests, claims and publishing.

0.6 The two kinds of box

The main text is complete without either kind of box. Skip them freely.

UNDER THE HOOD: WHAT THIS BOX IS FOR

A box like this one names the part of the Sharpee platform that a Chord line becomes: a trait, a standard action, an event, a message. It is for readers who also write for the platform directly, or who like to know what is underneath. Nothing later in the book depends on it.

FROM ANOTHER SYSTEM: WHAT THIS BOX IS FOR

A box like this one compares Chord with Inform 10 and TADS 3, and with no other system. It is for authors who already think in one of them. Each statement it makes about either system was checked against that system's own documentation.

0.7 Conventions

Lines of Chord appear in blocks like this one.

```
create the Lodge, a room
```

Commands for your terminal appear in their own blocks, and so does what the story prints. In a block of story output, a line that begins with `>` is a command the player typed.

```
> take lamp
You take it.
```

A block of Chord is sometimes a whole declaration and sometimes the few lines of one that the chapter is talking about. The prose says which lines are new, and the companion folder for the chapter always has the whole story.

In a story file the end of a line means something, and some lines are longer than a printed page is wide. When the page has to break such a line, the continuation begins with the mark `↪`, set under the start of the line it continues.

```
  description: Thirty years after the magic went out of the world,
  ↪ a lamp burns blue.
```

Type a marked row on the same line as the row above it, with one space where the page broke it, and leave the mark out. On a screen the line is not broken and there is no mark. Story output is prose, and the page wraps it as it wraps any paragraph.

0.8 Build and try it

Most chapters end with a section called “Build and try it”, which gives the `sharpee` commands that compile and play the chapter’s story in a

terminal. Chord Writer, the editor made for Chord, does the same work from one window. The story file is the same in both, and chapter 4 introduces the two.

0.9 The companion folders

The Relighting as it stands at the end of each chapter is on the book's website, in a folder named for the chapter: `chapter-05`, `chapter-06`, and so on. The folders are at this address:

https://sharpee.net/chord-manual/4.1.0/relighting/

Each one can be read in a browser, file by file, and each can be downloaded whole by adding `.zip` to its name:

https://sharpee.net/chord-manual/4.1.0/relighting/chapter-05.zip

A folder holds the story file and a test tree, which chapter 38 explains, and from Part VII on it holds the story's sounds, picture and stylesheet as well. Beside them are eight folders named `part-1` to `part-8`, each holding the story as it stands at the end of that part. No step in this book uses those.

You need the folders to follow the book. A chapter prints the lines it teaches. It does not print every line of the story, because a page of descriptions teaches nothing about Chord. Part I's story is short, and you type all of it. From chapter 5 on, each chapter's "Build and try it" begins by copying that chapter's `relighting.story` from its folder over your own, so that your story is the book's at the end of every chapter. The same section lists what changed, for the reader who would sooner make the changes by hand and take only the unprinted lines from the file.

0.10 When something is missing

A few chapters say plainly that a part of the language is not covered, because it could not be shown working in this edition. That is a

statement about the edition and not about your story. The language's reference material lists every form, and a later edition of this book will cover what this one could not.

PART I. A FIRST STORY

1 The shape of a story file

A Chord story is a plain text file whose name ends in `.story`. A small story needs nothing else: there is no project file beside it, and no second language in which the difficult parts are written. This chapter does not teach what any line means. It teaches what the page looks like, so that when later chapters hand you lines to type, you know where they go and how they sit.

Here is the whole of the story this book builds, as it stands at the end of Part I. It is called The Relighting. Look it over the way you would look over a page of sheet music before playing it.

```
## The Relighting, as it stands at the end of Part I: a header, a
↪ prologue,
## one room, and the person the player will be.

story
  title: The Relighting
  id: relighting
  story-version: 0.1.0
  authors:
    The Sharpee Project
  description: Thirty years after the magic went out of the world,
  ↪ a lamp burns blue.
  prologue:
    You were here for one term, thirty years ago, and you would
    ↪ have said
    you remembered none of it. Then last week a candle leaned
    ↪ toward your
    hand, and you started walking.

create the Lodge, a room

  The porter's lodge is one small room built into the thickness of
  ↪ the
  gate arch. A chair, a kettle and thirty years of newspapers have
  ↪ filled
  most of it. On the sill a lamp is burning with a steady blue
  ↪ flame.

create Ansel
  a person
  playable
  starts in the Lodge

  You were a student here once, for a term. You have the same
  ↪ hands and
  not much else.

before the game starts
  change the player to Ansel
end before
```

You can see the shape before you know the words. The file is a list of blocks. Each block begins with a line at the left margin, and the lines that

belong to it are indented beneath it. This file has four: the `story` block, two `create` blocks, and the `before the game starts` block. The two lines at the top that begin with `##` are a comment, which is a note to yourself that the compiler ignores.

FROM ANOTHER SYSTEM: LAYOUT

In Inform 10 the source is sentences and paragraphs, written to read as English. In TADS 3, braces mark the blocks, semicolons end the statements, and a line break means nothing. Chord is closer to a typed outline. A block is its header line and whatever is indented under it. There are no braces and no closing punctuation, and the indentation must be spaces. The compiler refuses a tab with the error `lex.tab-indent`, so set your editor to insert spaces before you begin.

UNDER THE HOOD: STORY IR · `SHARPEE COMPOSE`

The compiler does not run your file. It reads the file and writes a description of the story called the Story IR, which is a JSON document, and the Sharpee platform builds the world from that description. The command `sharpee compose relighting.story -o relighting.ir.json` writes it where you can read it. You do not have the `sharpee` command yet. Chapter 4 installs it, and runs this command at its end.

1.1 One line, one thing

Chord is read a line at a time. Each line says one thing, and the end of the line is the end of what it says. Nothing marks the end of a line except the line break: there is no full stop to remember and no semicolon to forget.

Look at the `create Ansel` block. It has a header line and then three short lines, `a person`, `playable` and `starts in the Lodge`. Each of the three is a separate fact about Ansel, and each has a line to itself.

Prose is the exception, and it is a comfortable one. The prologue and the two descriptions are running text, and running text may wrap wherever you like. The compiler joins the lines of a paragraph with single spaces, so the player sees one flowing paragraph however you broke it in the file.

1.2 Indentation says what belongs to what

A line that is indented further than the line above it belongs to that line. In the `story` block, `title:` and `authors:` belong to `story`, and the name `The Sharpee Project` belongs to `authors:` because it is indented further again. When the indentation returns to the left margin, the block is over.

This book indents by two spaces at each level. The compiler does not insist on two, but it does insist on spaces. A tab in the indentation is an error, and the compiler says so:

```
relighting.story:5:1 error [lex.tab-indent] Tabs are not allowed
in indentation — use spaces.
```

Most editors can be told to insert spaces when you press the Tab key. Set that before you type your first story, and you will never see this error.

1.3 How a block ends

Most blocks end when the indentation does. The `story` block ends where `create the Lodge` begins at the left margin, and nothing more is needed.

Some blocks hold a list of things to do, and those close with a line of their own. The last block in the file is one of them:

```
before the game starts
  change the player to Ansel
end before
```

The closing line is the word `end` followed by a word from the block's header. The chapters that introduce each kind of block tell you whether it needs one. If you leave out a closing line that is required, the compiler names the line it wanted and the line where the block began:

```
relighting.story:30:1 error [parse.unterminated-block] Missing
`end before` for the block opened at line 30.
```

1.4 Blank lines

Between blocks, a blank line is for your eyes. The compiler does not need one there, though this book always leaves one.

Inside a `create` block, a blank line does a job. It separates the short lines that say what a thing is from the paragraph that describes it. Look at the Lodge: the header line, a blank line, and then the description. Ansel has three short lines, a blank line, and then a description. Without the blank line the compiler would try to read the description as one more short fact, and it would fail.

A second blank line inside a description starts a second paragraph. The player sees the two paragraphs with a gap between them.

1.5 Comments

A comment is a note in the file that the story ignores. A comment line begins with two hash marks, and a run of comment lines makes one comment:

```
## The Relighting, as it stands at the end of Part I: a header, a
↪ prologue,
## one room, and the person the player will be.
```

Three rules govern comments, and all three are strict.

1. The `##` must be at the left margin. An indented `##` is not a comment.
2. A comment needs a blank line before it and a blank line after it. The top and the bottom of the file each count as a blank line.
3. A comment goes between blocks, never inside one.

There is no way to put a comment at the end of a line. If you write `title: The Relighting ## working title`, the title of your story is all of that, hash marks included. The same thing happens to an indented `##` under a description: it becomes a paragraph the player reads.

> **FROM ANOTHER SYSTEM: COMMENTS**
>
> Inform 10 writes a comment in square brackets, and TADS 3 has the two C forms, one that runs to the end of the line and one that brackets a region. Chord has one form with one place to put it. The reason is prose: a story file is mostly text the player will read, and the compiler takes text literally. A comment mark that worked inside a block would be a mark you could never print.

1.6 Words, numbers and quotation marks

The small pieces of a Chord line are few.

A **word** is letters, with digits, hyphens, apostrophes and underscores allowed inside it. Names are made of words, and a name may be several of them: `the Lodge`, `Ansel`, `The Sharpee Project`.

A **key** is a word that the language itself defines, followed by a colon, as in `title:` and `story-version:`. Where a key is made of two English words, it is always written with a hyphen and never with a space. Writing `story version:` is an error.

A **number** is digits, with full stops allowed inside it, so `0.1.0` is a number.

Quotation marks are rarer than you might expect. Text the player reads is not quoted. The title is not quoted, and neither is the prologue or any description. If you put quotation marks around the title, they become part of the title. Quotation marks do appear later, for a small number of things such as the name of a file or a topic the player asks about, and each chapter says so when it needs them.

> **FROM ANOTHER SYSTEM: QUOTED TEXT**
>
> In Inform 10 and in TADS 3, text the player reads is a quoted string, and everything outside the quotation marks is the language. Chord turns that around. Position decides what is prose: a paragraph under a `create` header is a description because of where it sits, and it needs no marks around it. The habit to unlearn is reaching for a quotation mark every time you write a sentence for the player.

1.7 What you have so far

You have not built anything yet, and that is deliberate. Type the story at the top of this chapter into a file called `relighting.story`, with spaces for indentation, and keep it. Make a new folder called `relighting` for it, with nothing else inside. The folder can be anywhere you keep your own work. Chapter 2 explains its first block, chapter 3 explains Ansel and the block that ends the file, and chapter 4 builds the story and plays it. The

Lodge waits for chapter 5, though it is already doing its work as the one place the story has.

2 The story header

Every story begins by saying what it is. In Chord that is the job of the `story` block, which this book calls the header. It holds the facts about the story as a whole: what it is called, who wrote it, which version this is, and the words the player reads before anything else happens.

Here is the header of The Relighting again.

```
story
  title: The Relighting
  id: relighting
  story-version: 0.1.0
  authors:
    The Sharpee Project
  description: Thirty years after the magic went out of the world,
  ↪ a lamp burns blue.
  prologue:
    You were here for one term, thirty years ago, and you would
    ↪ have said
    you remembered none of it. Then last week a candle leaned
    ↪ toward your
    hand, and you started walking.
```

The first line is the single word `story`. Everything indented beneath it is a **field**: a key, a colon, and a value. The header takes a fixed set of fields and no others, so you cannot invent one. This chapter covers the eight that describe the story. The rest belong to later chapters, most of them to Part VII, where the story is given its look in a browser.

2.1 The two fields every header needs

A header must have a `title:` and an `authors:` list. Nothing else is required. This is the smallest header the compiler accepts:

```
story
  title: The Relighting
  authors:
    The Sharpee Project
```

The **title** is everything after the colon, to the end of the line. It is not quoted, and it may contain any punctuation you like.

The **authors** are a list, even when there is only one. The key stands on a line by itself, and each name goes on its own line beneath it, indented one level further. A second author is a second line:

```
  authors:
    Hobb
    Tamsin Vey
```

You may not put a name on the same line as `authors:`. The compiler asks for the list form:

```
relighting.story:3:3 error [parse.header-inline-list] `authors:`
takes no inline name — list one name per line, indented under
`authors:`.
```

Leave either field out and the compiler names the one that is missing:

```
relighting.story:1:1 error [parse.story-title] The story block
requires a `title:` field.
```

2.2 What the player sees

When a story starts, the player is shown the title, the description if there is one, the authors and the version, and then the first room. With the smallest header above, and a first room cut down to one sentence so that the header's lines stand out, the opening looks like this. It is here to be read. Nothing in this chapter is built.

```
The Relighting
By The Sharpee Project
Story v0.0.0

Lodge
The porter's lodge is one small room.
```

With two authors the second line reads `By Hobb, Tamsin Vey`. The version reads `0.0.0` because the header did not give one. The next section fixes that.

2.3 Version, description and credits

`story-version:` is the version of your story, written as three numbers separated by full stops. It is yours to count however you please. This book gives The Relighting the version `0.1.0` at the end of Part I and raises the middle number with each Part.

`description:` is one line that says what the story is about. It appears under the title when the story starts, and again whenever the player types `about`.

`testers:` is a second list of names, written exactly like `authors:`. It is a place to record the people who played your story before it was finished.

With the version and the description added, typing `about` prints this:

```
> about
The Relighting
Version 0.1.0
By The Sharpee Project

Thirty years after the magic went out of the world, a lamp burns
blue.
```

2.4 The prologue

The **prologue** is the text the player reads first, before the title. It is where a story sets its scene.

```
prologue:
  You were here for one term, thirty years ago, and you would
  ↪ have said
  you remembered none of it. Then last week a candle leaned
  ↪ toward your
  hand, and you started walking.
```

The key stands alone on its line and the text follows beneath it, indented one level further. This is prose, so it wraps wherever you like and the player sees one paragraph. A blank line starts a second paragraph. The opening of The Relighting now reads:

```
You were here for one term, thirty years ago, and you would have
said you remembered none of it. Then last week a candle leaned
toward your hand, and you started walking.

The Relighting
Thirty years after the magic went out of the world, a lamp burns
blue.
By The Sharpee Project
Story v0.1.0

Lodge
The porter's lodge is one small room built into the thickness of
the gate arch. A chair, a kettle and thirty years of newspapers
have filled most of it. On the sill a lamp is burning with a
steady blue flame.
```

A short value may also sit on the same line as its key, which is how `description:` is written above. Both `description:` and `prologue:` accept either form.

There is one trap in the short form. If the value is a single word made of letters and hyphens, such as `prologue: opening-words`, the compiler

does not print that word. It takes it as the name of a phrase, which is a piece of text defined elsewhere in the file, and it reports an error if no phrase has that name. Chapter 20 teaches phrases. Until then, write your prologue as a sentence and the trap cannot catch you.

2.5 The two identifiers

`id:` is a short name for the story, written in lower-case letters with hyphens between the words. The player never sees it. The tools use it to name the things they make from your story, such as the folder that holds its tests and the folder that holds its browser build. Give every story an `id:`, and do not change it once you have chosen it.

`ifid:` is the story's Interactive Fiction Identifier, a long code that identifies one work of interactive fiction to archives and catalogues. You do not need to write one. The Relighting's header has none, and chapter 40 shows where the build puts the one it makes for you.

UNDER THE HOOD: THE HEADER IN THE STORY IR · SHARPEE COMPOSE

The header compiles to the `meta` section of the Story IR. The title is its own entry, and every other field you wrote is collected under `fields`:

```
"meta": {
  "title": "The Relighting",
  "fields": {
    "authors": [
      "The Sharpee Project"
    ],
    "testers": [],
    "themes": []
  }
}
```

That is the smallest header from earlier in this chapter. A field you did not write is simply absent, which is why the version was printed as `0.0.0`.

2.6 When a field is refused

Because the set of fields is fixed, a key the header does not know is an error, and the error lists every key it does know:

```
relighting.story:5:3 error [parse.header-unknown-field] Unknown
story-header field `genre:` — the header takes exactly: title,
authors, testers, ifid, id, story-version, prologue, description,
client, theme, template, themes, default-theme, storage-prefix,
publish-source, auto-assertion (plus states/score/use/on lines).
```

The words in parentheses at the end are not fields. They are other kinds of line that a header may hold, and each has its own chapter: `states` in chapter 13, `score` and `use` in chapter 18, and `on` in chapter 19.

Three keys from earlier versions of Chord are refused by name, and the error for each one tells you its replacement. `version:` is now `story-version:`, `blurb:` is now `description:`, and `by:` is now `authors:`.

The compiler is strict about which fields exist and relaxed about what you put in them. If you write `title:` twice, the second one wins and nothing warns you. An `id:` with spaces or capital letters in it is accepted as written. It is worth reading your header once more before you move on.

> **FROM ANOTHER SYSTEM: BIBLIOGRAPHIC DATA**
>
> Inform 10 lets the first line of the source give the title and the author, as an abbreviation, and sets the rest by sentences about the story, such as its description and its release number. TADS 3 keeps the same facts as properties of one object that describes the game. Chord's header is nearer to the TADS 3 arrangement, with one difference worth knowing early. The header is a closed form. You fill in the fields it has, and a fact it has no field for belongs somewhere else in the story.

2.7 What you have so far

Your `relighting.story` begins with a header that names the story, credits its author, gives it a version and a description, and sets the scene. The header says nothing yet about who the player is. That is the next chapter.

3 The player and the start

A story needs someone to be. In Chord the player is not a special object that exists before you write anything. The player is a role, and at the start of the story you hand that role to a person you have made. This chapter makes the person and hands over the role.

3.1 Making Ansel

The Relighting is played as Ansel, who was a student at Lanternhall for one term thirty years ago and has walked back this morning. Here is Ansel in full.

```
create Ansel
  a person
  playable
  starts in the Lodge

  You were a student here once, for a term. You have the same
  ↪ hands and
  not much else.
```

The header line, `create Ansel`, makes something and names it. Chapter 5 has much more to say about `create`, since everything in a story's world is made with it. For now it is enough to read the three short lines beneath the header, each of which states one fact.

`a person` says what kind of thing Ansel is. A person can carry things and wear things, and is someone in the story rather than something.

`playable` says that Ansel may hold the player role. Only a person can be playable. This line does not make Ansel the player. It makes Ansel a person who is allowed to be.

`starts in the Lodge` says where Ansel is when the story begins. The Lodge is the one room the story has so far.

After the blank line comes Ansel's description. The player reads it by typing `examine me`:

```
> examine me
You were a student here once, for a term. You have the same hands
and not much else.
```

Notice that the description is written to the player, as "you". Everything the player reads in The Relighting is in the second person and the present tense, which is the custom in interactive fiction, and you will be writing in that voice from here on. A description is optional. If you leave Ansel's out, the platform supplies its stock answer, which is "As good-looking as ever."

3.2 Handing over the role

Making a playable person is half of the job. The other half is the last block in the file.

```
before the game starts
  change the player to Ansel
end before
```

This is the **start block**. It runs once, before the first turn, and its job is to say who the player is. The line inside it, `change the player to Ansel`, gives the role to Ansel. When the story opens, the player is standing where Ansel starts, which is the Lodge.

The phrase `the player` deserves a moment. It does not name a person. It means whoever holds the role at the time. Later in this book you will write lines such as "when the player takes the lamp", and those lines will go on working if the role moves to someone else in the middle of the story. Chapter 31 does exactly that.

A story has exactly one start block, and the block must hand the role to someone. It may go anywhere in the file after the header. This book puts it last.

3.3 What the start block is not for

The start block is not the place to greet the player. It runs before there is a turn to carry any text, so it cannot print anything. The words the player reads first belong in the header's `prologue:` field, which chapter 2 described. Keep the two apart in your mind: the prologue is what the player reads at the start, and the start block is what the story arranges at the start.

3.4 The opening

Once the start block has run, the story does three things unasked, before the player has typed a word. It prints the prologue. It prints the title with the description, the authors and the version, which together are called the **banner**. Then it takes a first look at the room the player is standing in. These three together are the **opening**.

```
You were here for one term, thirty years ago, and you would have
said you remembered none of it. Then last week a candle leaned
toward your hand, and you started walking.

The Relighting
Thirty years after the magic went out of the world, a lamp burns
blue.
By The Sharpee Project
Story v0.1.0

Lodge
The porter's lodge is one small room built into the thickness of
the gate arch. A chair, a kettle and thirty years of newspapers
have filled most of it. On the sill a lamp is burning with a
steady blue flame.
```

The opening is not a turn. Nobody typed anything, so nothing that counts turns has counted one, and the first command the player types is turn 1. That will matter in chapter 19, where the story gets a clock.

Each of the three has a name, and a second block lets you say which of them run and in what order.

```
when the game starts
  show-prologue
  show-banner
  do-first-look
end when
```

This is the **opening block**. It opens with `when the game starts` and closes with `end when`, and between them it holds the names of the opening's three **routines**, each alone on its line. The block above is the order a story gets when it has no opening block at all, so adding it to your file changes nothing.

Change the order of the lines and the opening changes with them. Here the banner comes first, in the way of a book that puts its title page before its first paragraph.

```
when the game starts
  show-banner
  show-prologue
  do-first-look
end when
```

```
The Relighting
Thirty years after the magic went out of the world, a lamp burns
blue.
By The Sharpee Project
Story v0.1.0

You were here for one term, thirty years ago, and you would have
said you remembered none of it. Then last week a candle leaned
toward your hand, and you started walking.

Lodge
The porter's lodge is one small room built into the thickness of
the gate arch. A chair, a kettle and thirty years of newspapers
have filled most of it. On the sill a lamp is burning with a
steady blue flame.
```

A routine you leave out does not run. A block without `do-first-look` prints the prologue and the banner and then waits, and the player reads about the Lodge only when they look for themselves. A block with no lines in it prints nothing at all.

The block holds routines and nothing else. Each may appear once, and `show-prologue` may appear only in a story whose header has a `prologue:` field. The compiler refuses a routine written twice, and it refuses any other statement:

```
relighting.story:36:3 error [analysis.opening-routine-repeated]
`show-banner` is already in this block. An opening runs each
routine once; take this line out.
```

```
relighting.story:36:3 error [parse.opening-statement] `phrase
hello` cannot go in a `when the game starts` block. The block
holds only the opening's routines, each alone on its line: `show-
prologue`, `show-banner`, `do-first-look`.
```

A story has at most one opening block, and like the start block it may go anywhere in the file after the header. The two are easy to tell apart by their first words. `before the game starts` arranges the world and

prints nothing. `when the game starts` prints the opening and arranges nothing.

The Relighting keeps the usual order, so this book's story has no opening block, and you need not add one.

3.5 When it goes wrong

There are four mistakes to make here, and the compiler catches three of them.

If you leave the start block out, the story has no player:

```
relighting.story:4:1 error [analysis.start-block-missing] This
story never says who the player is. Add a `before the game starts`
block assigning the role: `change the player to <character>`.
```

If you hand the role to a person who is not `playable`, the compiler tells you which line to add:

```
relighting.story:30:3 error [analysis.player-target-not-playable]
`ansel` is not `playable` — add `playable` to its create block to
let the player role move to it.
```

If you write two start blocks, the compiler asks you to merge them.

The fourth mistake is leaving out `starts in the Lodge`. The compiler accepts that story, and the mistake shows itself only when you try to play:

```
Error loading story: Story player "Ansel" (ansel) is not placed in
the world — an unplaced player character is nowhere to play.
```

So give every playable person a place to start, whatever the compiler says.

UNDER THE HOOD: THE PLAYER ROLE · `SHARPEE COMPOSE`

In the Story IR, each thing you create is an entry in `entities`, and `playable` is a flag on that entry:

```
"name": "Ansel",
"article": null,
"aka": [],
"isPlayable": true,
```

The platform never begins with a player of its own. When it loads a story it runs the start block, finds the person who now holds the role, and refuses to start if that person has no place in the world. That refusal is the error you have just read.

FROM ANOTHER SYSTEM: THE PLAYER CHARACTER

In Inform 10 a story has a player without your asking for one, and the first thing an author says about the player is where they are. TADS 3 has you define the player character as an object and name it in the game's main object. Chord has no default player at all. A story with no start block does not compile, which means the question of who the player is always has an answer you wrote.

3.6 What you have so far

The whole of `relighting.story` is now explained except for the room in the middle, and you have typed all of it. The next chapter builds it, plays it, and breaks it on purpose.

4 Build, play, read an error

You have a story file and you understand every block in it but one. This chapter turns it into something you can play. It also breaks the file in several ways on purpose, because the most useful skill in this book is reading what the compiler says when something is wrong.

There are two ways to work with a Chord story. Chord Writer is an application with an editor, a play pane and a testing pane in one window. The `sharpee` command does the same work from a terminal. This book gives both wherever a step is something you do rather than something you write, under the heading “Build and try it”. The story file is the same either way. This chapter is steps from beginning to end, so it has no such heading. From chapter 5 on, each chapter has one near its end.

4.1 Installing the command

The `sharpee` command needs Node.js, version 18 or later. If you do not have it, install it from nodejs.org first. Then install the command itself:

```
npm install -g @sharpee/devkit
```

A recent npm may end the install with a warning that one package had its install script blocked. The command works without that script, and you can leave the warning alone.

Check that the command is there by asking for its version:

```
sharpee --version
```

```
Sharpee 5.7.0 · Chord 4.1.0
```

The first number is the version of the Sharpee platform, which runs your story. The second is the version of the Chord language, which is what you write it in.

4.2 A folder for the story

Chapter 1 had you make a folder called `relighting` and put `relighting.story` in it. A story's folder is called its **project**, and a project holds exactly one `.story` file. Everything in this book is typed from inside that folder.

4.3 Checking the story

The first thing to do with a story file is to ask whether the compiler accepts it.

```
sharpee compose relighting.story --check
```

```
compose: Chord 4.1.0 — relighting.story is gate-clean (--check: IR
not emitted)
```

The word to look for is **gate-clean**. The compiler puts a story through a series of checks, which it calls gates, and a story that passes every one of them is gate-clean. A story that is not gate-clean cannot be played.

`compose` is the compiler's name, and `--check` tells it to run the gates and stop. Run this command often. It takes a moment, and it is far easier to find a mistake among the five lines you have just typed than among fifty.

4.4 Playing the story

To play the story, tell `sharpee` to play the project you are standing in. The full stop means "this folder".

```
sharpee play .
```

```
Loading story from: /home/ansel/relighting

--- Play Mode ---  (/quit or /q to exit)

You were here for one term, thirty years ago, and you would have
said you remembered none of it. Then last week a candle leaned
toward your hand, and you started walking.

The Relighting
Thirty years after the magic went out of the world, a lamp burns
blue.
By The Sharpee Project
Story v0.1.0

Lodge
The porter's lodge is one small room built into the thickness of
the gate arch. A chair, a kettle and thirty years of newspapers
have filled most of it. On the sill a lamp is burning with a
steady blue flame.

>
```

The first line names the folder the story was loaded from, and yours will name your own. After it come your prologue, then your header, then your room. The `>` is the prompt, where the player types. Try a few commands.

```
> examine me
You were a student here once, for a term. You have the same hands
and not much else.

> examine lamp
You can't see any such thing.

> north
There are no obvious exits.

> wait
Time passes...

> xyzzy
I don't understand that.
```

Two of those answers are worth a second look, because you wrote neither of them.

“You can’t see any such thing” is the story telling the truth. The Lodge’s description mentions a lamp, but a description is only words. There is no lamp in the story until you create one, and chapter 6 does. Every noun you put in a description is a promise to the player that it can be examined, and for now The Relighting is breaking several such promises in one paragraph.

“There are no obvious exits” and “Time passes…” are answers that came with the platform. You did not teach the story what going north means, or what waiting is. A Chord story begins with several dozen ordinary actions already understood, and most of this book is about adding to them, not building them.

Type `/quit` to leave.

4.5 Reading an error

Now break the story. Open `relighting.story`, find the line `playable` under `create Ansel`, and remove one letter so that it reads `playble`. Check the story again.

```
sharpee compose relighting.story --check
```

```
relighting.story:31:3 error [analysis.player-target-not-playable]
`ansel` is not `playable` — add `playable` to its create block to
let the player role move to it.
relighting.story:24:3 error [analysis.trait-not-declared] Trait
`playble` is not declared (`define trait playble`) and is not a v1
adjective.
compose: relighting.story failed the load-time gates (2 error(s))
```

Each line the compiler prints about your story is called a **diagnostic**, and every diagnostic has the same five parts.

1. **Where.** `relighting.story:24:3` is the file, the line and the column. Line 24, column 3 is the first letter of `playble`.
2. **How serious.** `error` means the story cannot be played until this is fixed. The other word you will see here is `warning`, which is described below.
3. **The code.** `[analysis.trait-not-declared]` is the diagnostic's name. The part before the full stop says which stage of the compiler found the problem: `lex` reads characters, `parse` reads lines and blocks, and `analysis` checks that what you wrote makes sense together. The code never changes, so it is the thing to search for when a message puzzles you.
4. **The message.** This says what is wrong in a sentence.
5. **The remedy.** Most messages end by telling you what to write instead.

Now read the two diagnostics as a pair. The second one is the mistake itself: the compiler does not know the word `playble`. The first is a consequence of it: because that line was not understood, Ansel is not

playable, and so the start block is handing the role to someone who cannot hold it. One wrong letter produced two errors, and the one printed first is not the one to fix.

This happens often, and it gives the rule for reading compiler output. Read all of it before you change anything, look for the diagnostic that points at something you typed wrongly, and fix that one. Then check again. The others frequently vanish.

Put the letter back before you go on.

4.6 When the compiler guesses

Some diagnostics go further than a remedy and offer a guess. Change `starts in the Lodge` to `starts in the Loge` and check again.

```
relighting.story:25:13 error [analysis.unknown-entity] No entity
named `Loge` — did you mean `lodge`?.
compose: relighting.story failed the load-time gates (1 error(s))
```

An **entity** is the compiler's word for anything you make with `create`. The guess is printed in lower case because the compiler compares names without regard to capitals. You may write `the Lodge` or `the lodge` in your own lines, and they are the same room. Put the letter back before you go on.

4.7 Warnings

A warning is a diagnostic that does not stop the story. Delete the Lodge's description, both the paragraph and the blank line above it, so that the header line has nothing under it, and check.

```
relighting.story:16:8 warning [analysis.room-no-description] `the
Lodge` has no description, so LOOK there shows only its name.
compose: Chord 4.1.0 — relighting.story is gate-clean (--check: IR
not emitted)
```

The story is still gate-clean and it will still play. The warning is the compiler pointing at something that is legal and probably not what you meant. Treat a warning as a question. Sometimes the answer is that you did mean it.

Restore the description, check once more, and make sure you are back to a single gate-clean line.

4.8 Two mistakes with the folder

Two errors come from the project and not from the story, and both are easy to make in your first hour.

The `play` command takes a folder, not a file. If you give it the file, it says so in its own way:

```
play: 'relighting.story' is neither a directory nor a registered
story — run `sharpee register <location>`, or run `sharpee play`
from the project directory
```

A project has one story. If you keep a second `.story` file in the folder, a copy you made before an experiment, for instance, `play` refuses to choose between them:

```
Error loading story: project has 2 .story files (notes.story,
relighting.story) — a project has exactly one; remove or rename
the others
```

Keep your experiments in another folder, or give the copy a different ending.

UNDER THE HOOD: WHAT `compose` WRITES · `sharpee compose`

Without `--check`, the compiler writes the Story IR. Give it a file to write to:

```
sharpee compose relighting.story -o relighting.ir.json
```

```
compose: Chord 4.1.0 — relighting.story loaded — 2 entities,
0 trait(s), 0 action(s)
compose: IR written to relighting.ir.json
```

The two entities are the Lodge and Ansel. The traits and actions it counts are ones a story defines for itself, which Part IV teaches, and The Relighting has none yet. The file is JSON and is safe to open and read. Nothing in this book asks you to edit it, and `sharpee play` does not need it: it compiles the story afresh each time.

FROM ANOTHER SYSTEM: THE COMPILE STEP

In Inform 10 you press Go and the application compiles and runs the story in one motion. In TADS 3 you compile a project into an image file and run that in an interpreter. Chord has the same two steps, but nothing you keep comes out of them at this stage. There is no story file to pass to an interpreter. The `.story` text is the story, `play` compiles it in memory each time, and a thing you can give to a player is made only when you publish, in chapter 40.

4.9 What you have so far

You can check a story, play it, and read what the compiler says about it. The Relighting is one room with nothing in it and nowhere to go. Part II gives it a world.

PART II. THE WORLD

5 Rooms and exits

One room is a place to stand. A world is rooms with ways between them. In this chapter The Relighting grows from the Lodge to all eight rooms of Lanternhall, and by the end of it you can walk from the gate to the roof.

5.1 What `create` does

Everything in a story's world is made with a `create` block. The header line gives the thing its name, and the first fact beneath it says what kind of thing it is.

```
create the Lodge
  a room
```

The first fact is so nearly universal that Chord lets you fold it onto the header line, after a comma. This book writes rooms that way from here on.

```
create the Lodge, a room
```

Every `create` block names exactly one kind, and the kind comes first. If you leave it out, as in a block that begins `create the Refectory` and never says what the Refectory is, the compiler lists the kinds there are:

```
relighting.story:54:1 error [analysis.missing-kind-noun]
`Refectory` names no kind. Begin its block with one: `a room`, `a
door`, `a person`, `a container`, `a supporter`, `a region`, or `a
thing` for anything that is not a room, door, person, container,
supporter or region.
```

That list is most of Part II. You have met `a room` and `a person`. Chapter 6 covers `a thing`, chapter 7 `a container` and `a supporter`, chapter 8 `a door` and chapter 11 `a region`.

The name is yours to choose. Write it as you want the player to read it, with `the` in front if it is the kind of name that takes one. The player sees the name of a room as the heading over its description.

5.2 Exits

An **exit** is a way out of a room. It is one line in the room's block: a direction, the word `to`, and the room it leads to.

```
create the Lodge, a room
  north to the Gate Court

  The porter's lodge is one small room built into the thickness of
  ↪ the
  gate arch. A chair, a kettle and thirty years of newspapers have
  ↪ filled
  most of it. On the sill a lamp is burning with a steady blue
  ↪ flame. The
  door to the court is north.
```

The exit line goes with the short facts, above the blank line. The room it names does not have to be created yet. It only has to be created somewhere in the file.

There are twelve directions: `north`, `south`, `east` and `west`, the four between them (`northeast`, `northwest`, `southeast`, `southwest`), and `up`, `down`, `in` and `out`.

Notice the sentence added to the end of the Lodge's description: "The door to the court is north." The story does not list a room's exits for the player. Telling the player which ways lead out is the description's job, and a room whose description forgets to do it is a room the player cannot leave except by trying every direction.

5.3 The way back is free

An exit works in both directions. Because the Lodge has `north to the Gate Court`, the player can also go south from the Gate Court to the Lodge. You do not write the second line.

You may write it if it makes the file easier to read, as long as it agrees. The Gate Court does so, and uses the second way of writing exits while it is about it:

```
create the Gate Court, a room
  exits: south to the Lodge, north to the Cloister
```

The `exits:` line holds several exits on one line, separated by commas. It means exactly what the same exits would mean on separate lines. Use whichever reads better for the room in hand.

What you may not do is contradict yourself. A room has one exit in each direction, and the free way back counts. Suppose the Den goes `up to the Attic`, and the Attic then says `down to the Cellar`. The compiler objects, and its message is one of the most helpful in the language:

```
relighting.story:12:3 error [analysis.duplicate-exit] `Attic`'s
`down` exit leads to `Cellar`, but `Den`'s `up to the Attic` (line
7) also makes `down` from `Attic` lead back to `Den`. If that line
should not lead back, write it `up to the Attic, one-way`.
```

5.4 One-way exits

That message names the remedy. An exit followed by `, one-way` has no way back.

```
create the Den, a room
  up to the Attic, one-way
```

The player can climb from the Den to the Attic and cannot climb down again by that route. If the Attic has no other exit, going down from it prints the platform's line for a room with no ways out at all:

```
> down
There are no obvious exits.
```

On an `exits:` line the words go before the colon and apply to every exit on the line, as in `exits, one-way: up to the Attic`.

The Relighting has no one-way exits yet. Use them sparingly. A player who cannot go back the way they came should be able to tell why from the prose.

5.5 The rest of Lanternhall

Here are the six rooms that complete the map. Read the exit lines against the descriptions, and notice that only one room of each connected pair declares the way between them.

```
create the Cloister, a room
  east to the Refectory
  west to the Library
  north to the Bell Stair
  down to the Undercroft

  A covered walk around a garth gone to nettles. The refectory is
  ↪ east,
  the library west, and the foot of the bell stair north. Worn
  ↪ steps go
  down under the hall.

create the Refectory, a room

  Three long tables and a hearth you could stable a horse in. The
  benches are stacked against the wall, where someone left them
  ↪ for the
  holidays. The cloister is back to the west.

create the Library, a room

  The reading room of Lanternhall, with its desks in rows and its
  galleries of shelves. The stacks run back to the north, out of
  ↪ the
  light. The cloister is east.

create the Undercroft, a room

  Low vaults on squat pillars, running off under the whole of the
  ↪ hall.
  The air is cold and smells of stone. The steps go back up.

create the Bell Stair, a room
  up to the Lantern Gallery

  The foot of the tower. A stair winds up around the well where
  ↪ the bell
  rope hangs. The cloister is south.

create the Lantern Gallery, a room

  The roof of the tower, open on every side. The Great Lantern
  ↪ stands in
```

```
  the middle of it on a stone drum, taller than you are, its glass
  thirty years dirty. The stair goes back down.
```

The Cloister is the hub, so it carries four exit lines and the rooms around it carry none. That is a matter of taste. Putting the exits on the hub keeps the whole map of the Hall in one place in the file.

5.6 The first visit

A room may say something different the first time the player walks into it. The Gate Court does.

```
create the Gate Court, a room
  exits: south to the Lodge, north to the Cloister

  A square of worn flagstones between the gate and the cloister
  ↪ arch.
  Grass has come up through the joints and been left to get on
  ↪ with it.
  Over the gate, a line of cut letters says that the craft needs a
  ↪ light
  to come home to.

  first time
    You crossed this court every morning for a term, late, with
    ↪ your
    books under your arm. It was larger then. The cloister arch is
    still north of you.
```

The `first time` block comes after the description, and its text is indented one level further beneath it. On the first visit the player reads that text in place of the description. On every later visit, and whenever they type `look`, they read the description.

```
> north
Gate Court
You crossed this court every morning for a term, late, with your
books under your arm. It was larger then. The cloister arch is
still north of you.

> look
Gate Court
A square of worn flagstones between the gate and the cloister
arch. Grass has come up through the joints and been left to get on
with it. Over the gate, a line of cut letters says that the craft
needs a light to come home to.
```

Because the first-visit text replaces the description, anything the player needs from the description has to be in both. That is why the first-visit text ends by saying where the cloister arch is. Without that sentence, a player walking into the Gate Court for the first time would be told nothing about the way north until they typed `look`.

5.7 A heading of its own

The name you give a room in its `create` line is the name you use for it everywhere in the file, so a short one is convenient. It is also, by default, the heading the player reads, and there you may want something fuller. A `room name` block gives the room a heading that is separate from its name.

```
create the Lodge, a room
  north to the Gate Court

  room name:
    The Porter's Lodge
```

You go on writing `the Lodge` in your own lines, and the player reads:

```
> south
The Porter's Lodge
The porter's lodge is one small room built into the thickness of
the gate arch. A chair, a kettle and thirty years of newspapers
have filled most of it. On the sill a lamp is burning with a
steady blue flame. The door to the court is north.
```

Without the block, the heading is the room's name with its `the` removed, which is why the other rooms read "Gate Court" and "Cloister".

5.8 Build and try it

From this chapter on, the story is too long to type from the page, and the chapters do not print every line of it. Copy `relighting.story` from the companion folder `chapter-05` over your own file. It is the story with this chapter's changes made, and it holds the descriptions the chapter did not print in full. "How to read this book" says where the companion folders are. To make the changes by hand instead, work from this list and take the rest from that file: add the seven new rooms and the changes to the Lodge, and raise `story-version:` to `0.2.0`, since this is the first chapter of Part II. Check the story, then play it.

```
sharpee compose relighting.story --check
sharpee play .
```

Walk the whole map. From the Lodge, the commands `north`, `north`, `north` and `up` take you to the roof.

```
> north
Bell Stair
The foot of the tower. A stair winds up around the well where the
bell rope hangs. The cloister is south.

> up
Lantern Gallery
The roof of the tower, open on every side. The Great Lantern
stands in the middle of it on a stone drum, taller than you are,
its glass thirty years dirty. The stair goes back down.
```

Then try a direction that has no exit. On the roof, where the only way is down, try `east`.

```
> east
You can't go that way.
```

If a room you expected is unreachable, the usual cause is a misspelled name on an exit line. The compiler catches that, and guesses what you meant:

```
relighting.story:30:39 error [analysis.unknown-entity] No entity
named `Cloyster` — did you mean `cloister`?.
```

UNDER THE HOOD: THE GOING ACTION · `IF.ACTION.GOING`

Typing a direction runs one of the platform's standard actions, called `going`. An exit line gives that action somewhere to go. Each of the twelve directions is a word the action already understands, along with its short form, such as `n` for `north`. You did not define any of that. The directions are a closed list in the language. Chapter 12 shows the ways an exit can be closed, hidden or made dangerous, and all of them work by giving `going` a reason to refuse.

FROM ANOTHER SYSTEM: THE MAP

Inform 10 builds its map from sentences such as “The Gate Court is north of the Lodge”, and it also makes the way back for you unless you say otherwise. TADS 3 gives each room a property for each direction, and there you write both directions yourself. Chord sits with Inform 10 on this point: one line makes a two-way connection, and you ask for one-way when you want it. Where Chord differs from both is that a room’s exits are lines inside that room’s own block. There is no separate place where the map is described.

5.9 What you have so far

Lanternhall has its eight rooms, and every one of them can be reached. They are empty. The descriptions promise a lamp, a kettle, a gate, a hearth and a Great Lantern, and not one of them exists. The next chapter starts keeping those promises.

6 Things

The rooms of Lanternhall describe a lamp, a kettle, a chair, a gate and a hearth, and the player can touch none of them. This chapter makes them. It also gives Ansel something to carry and puts the first other person in the story.

6.1 A thing and where it is

Anything in the world that is not a room, a door, a person, a container, a supporter or a region is `a thing`. A thing needs two facts: its kind, and its place.

```
create the kettle
  a thing
  in the Lodge

  A black iron kettle, still warm.
```

The line `in the Lodge` is the thing's **placement**. It is the word `in`, an article, and the name of a room. The paragraph after the blank line is the thing's description, which the player reads by examining it.

That is all a thing needs in order to be seen, examined, picked up, carried about and put down. You wrote none of those abilities. They come with being a thing.

```
You can see a kettle, a hand lamp, and Hobb here.

> examine lamp
A tin hand lamp with a wire handle. Its wick is dry and its glass
is clean, which is Hobb all over.

> take lamp
Taken.
```

The line beginning “You can see” is the story’s own. After a room’s description it lists the things in the room that the description has not accounted for, and it chooses “a” or “an” for each one. The kettle and the hand lamp are listed because they are loose things a player might pick up.

A description is optional here too. A thing without one answers an examination with a stock line, which for the kettle would be “The kettle is just a kettle.”

6.2 Things that stay put

The Lodge’s description has already told the player about the chair and the newspapers. They should not be listed a second time, and nobody should walk off with the porter’s chair. Both problems have one answer, which is the word `scenery`.

```
create the chair
  a thing, scenery
  in the Lodge

  A wing chair that has taken the shape of one man.
```

`scenery` is the first of the **trait words**. A trait word is an adjective that follows the kind, after a comma, and changes what the thing is like. This one says two things about the chair: it is part of the room’s description, so the room does not list it, and it cannot be taken.

```
> take chair
The chair is fixed in place.
```

Scenery is how you keep the promises a room description makes. Read each of your descriptions as a player would, note every noun in it, and make a scenery thing for each one that a player might reasonably examine. The Relighting now does this for the blue flame on the sill, the chair, the newspapers, the gate, the hearth in the Refectory and the Great Lantern on the roof.

6.3 More than one, and things with names

Two more trait words are about grammar, the player's and the story's.

`plural` says that the thing's name is a plural noun. The story then leaves out the "a" it would otherwise put in front of the name, so that a room lists "newspapers" and not "a newspapers".

```
create the newspapers
  aka papers
  a thing, scenery, plural
  in the Lodge

  The top one is from last week. The bottom one reports the
  ↪ Dimming, in
  the small type newspapers use for things they do not yet
  ↪ believe.
```

`proper` says that the name is a proper name, one that takes no article. It is most often needed for people.

```
create Hobb
  a person, proper
  in the Lodge

  The porter of Lanternhall, who was old when you were a student
  ↪ and has
  not troubled to change since.
```

Without `proper` the Lodge would report that you can see “a Hobb” here. With it, he is listed as Hobb, and the story’s own messages use his name correctly:

```
> take hobb
You can't take Hobb.
```

Hobb is a person, not a thing, and people are placed the same way things are. He does nothing yet except stand in the Lodge and be examined. Part V is his.

6.4 Other words for a thing

The player will not always use your name for a thing. An `aka` line lists the other words they might use, separated by commas. It goes directly under the header, above the kind.

```
create the gate
  aka gates, arch, letters, motto
  a thing, scenery
  in the Lodge and the Gate Court
```

Now `examine arch` and `examine motto` both find the gate. Give a thing an `aka` for every noun your own prose uses about it. The Gate Court’s description speaks of “a line of cut letters”, so `letters` is on the list.

Be careful with the opposite problem. When two things in the same place can be meant by the same word, the story has to ask which one the player

means, and it does so every time. If the Lodge held both a brass key and an iron key, then `take key` would be answered with a question. The cure is to choose names that do not share their last word, or to make sure the two things are never in the same place. This book called the lamp on the sill `the blue flame` for exactly that reason: there is a hand lamp in the same room, and the word `lamp` should mean only one of them.

6.5 A thing in several rooms

Look again at the gate's placement.

```
   in the Lodge and the Gate Court
```

The gate can be seen from both rooms, so it is placed in both. A thing placed in more than one room is called a **backdrop**. With three or more rooms the list is written as English writes it, `in the A, the B and the C`.

A backdrop must be `scenery`. A thing the player could pick up cannot be in two places, and the compiler explains as much if you forget:

```
relighting.story:134:3 error [analysis.backdrop-not-scenery]
`gate` is placed in several rooms — only scenery can be in more
than one place (a thing the player could take cannot). Add
`scenery`, or name one room.
```

Backdrops are for the sky, the river, the distant tower, and anything else that a player in several rooms would expect to be able to look at.

6.6 What Ansel carries

A person can begin the story holding things. The line `carries`, in the person's own block, names one.

```
create the wax taper
  aka taper
  a thing

  Half a wax taper. Last week you lit it to read by, and its flame
  leaned toward your hand like a cat. You have carried it since.

create Ansel
  a person
  playable
  starts in the Lodge
  carries the wax taper
```

The taper has no placement line of its own. Ansel's `carries` line places it. Write one `carries` line for each thing.

```
> inventory
You are carrying:

a wax taper
```

A thing with no placement at all, and that nobody carries, is not an error. It exists and is nowhere. That is useful later, for things that enter the story part of the way through, and chapter 15 shows how to bring one on.

6.7 Hidden things

One more trait word belongs here. A thing that is **`concealed`** is in its room but cannot be seen, examined or taken until the player finds it by searching.

```
create the spectacles
  aka glasses
  a thing, plural, concealed
  in the Lodge

  Wire spectacles with one arm mended in thread. Hobb has been
  ↪ looking
  for them since the spring.
```

```
> examine spectacles
You can't see any such thing.

> search
Hidden here, you discover: spectacles.

> take glasses
Taken.
```

Once found, a concealed thing is an ordinary thing. Use this sparingly as well. A player has no reason to type `search` unless your prose gives them one, and thirty years of newspapers is the Lodge's hint.

The companion word is **`hiding-spot`**, which marks a thing the player can hide behind. The Relighting has no use for one, so here is a curtain from another story:

```
create the curtain
  a thing, scenery, hiding-spot
  in the Den

  A heavy curtain.
```

```
> hide behind curtain
You slip behind the curtain.
```

6.8 Build and try it

Copy `relighting.story` from the companion folder `chapter-06` over your own file. It is the story with this chapter's changes made, and it holds the things and descriptions the chapter did not print in full, the hand lamp, the blue flame, the hearth and the Great Lantern among them. Check it and play it.

```
sharpee compose relighting.story --check
sharpee play .
```

Take the hand lamp, carry it north, and put it down in the Gate Court.

```
> drop lamp
Dropped.

> look
Gate Court
A square of worn flagstones between the gate and the cloister
arch. Grass has come up through the joints and been left to get on
with it. Over the gate, a line of cut letters says that the craft
needs a light to come home to.

You can see a hand lamp here.
```

Then examine everything a description mentions, in every room. Where the answer is "You can't see any such thing", you have found a promise the story is not yet keeping. Some of those are deliberate, because the Library's desks and the bell rope have chapters of their own coming. Others stay unkept in The Relighting, which makes scenery only of what the game uses. Nothing in this step needs fixing. In a story of your own, each of them is a line to write.

UNDER THE HOOD: TRAITS · SHARPEE COMPOSE

A kind and its trait words become **traits** on the platform, which are the units of ability it builds everything from. `scenery` is the scenery trait, and the standard `taking` action refuses anything that has it, which is where "fixed in place" comes from. `a person` brings the actor trait, and `taking` refuses that as well, with a different sentence. The words you write are a closed list. You cannot invent a trait word by typing one, and a word the compiler does not know is reported as `analysis.trait-not-declared`. Chapter 24 shows how to declare traits of your own.

FROM ANOTHER SYSTEM: SCENERY AND BACKDROPS

Both ideas will be familiar. Inform 10 has scenery as a property of a thing and a backdrop as a kind of its own, placed in several rooms or in a region. TADS 3 has a class for a thing that must exist in more than one place at a time. In Chord, scenery is a trait word, and a backdrop is not a kind at all. It is simply scenery whose placement names more than one room. The other difference is that Chord does not let a thing be portable by declaration and immovable by rule: `scenery` is the one switch, and it covers both the listing and the taking.

6.9 What you have so far

The Lodge has a porter, his chair, his papers, his kettle and two lamps. Ansel is carrying a taper. The gate can be seen from both sides, and the hearth and the Great Lantern can be examined. Nothing yet can be opened, and nothing can be put on or in anything else. That is the next chapter.

7 Containers and supporters

A thing can be picked up and put down. This chapter adds the two kinds that other things can be put into and onto, and with them the first puzzle-shaped object in The Relighting: a closed crate with something in it.

7.1 Two kinds that hold

`a container` is a thing with an inside. Things go in it.

`a supporter` is a thing with a top. Things go on it.

Each is a kind, so it takes the place of `a thing` on the first line of the block. A container is already a kind of thing, and you do not write both.

```
create the reading desk
  aka desk
  a supporter, scenery
  in the Library

  The first desk of the first row, with a slope for a book and a
  ↪ rail to
  keep it there.

create the grimoire
  aka book
  a thing
  on the reading desk

  A book as long as your forearm, chained to nothing any more. Its
  ↪ pages
  are blank, or were. Here and there a line of ink has come back,
  ↪ as
  frost comes onto a window.
```

The grimoire's placement is new. Chapter 6 placed things with `in the` followed by a room. A thing may equally be `in the` a container or `on the` a supporter. The grimoire is `on the reading desk` because the desk is a supporter. Take care to use the word that fits the kind, because the compiler does not check it for you.

The player needs no new commands. Putting, taking and looking all understand containers and supporters already.

```
> west
Library
The reading room of Lanternhall, with its desks in rows and its
galleries of shelves. The stacks run back to the north, out of the
light. The cloister is east.

On the reading desk you see a grimoire.

> take book
You take the grimoire from the reading desk.

> put book on desk
You put the grimoire on the reading desk.
```

The desk is scenery, so the room does not list it. What is on the desk is listed all the same, in a sentence of its own. Scenery hides the holder and not what it holds.

7.2 Opening and closing

A container may be given a lid. The trait word is `openable`, and beside it you say how the container begins.

```
create the crate
  a container, openable, starts closed
  in the Undercroft
  holds 1

  A deal crate packed with straw, stencilled FRAGILE on every
  ↪ face.

create the lens
  a thing
  in the crate

  A disc of glass as wide as a cartwheel, ground in rings. A crack
  ↪ runs
  across it from edge to edge.
```

`starts closed` and `starts open` are the two choices. What is inside a closed container cannot be seen or touched, and for the player it does not exist until the lid comes off.

```
> take lens
You can't see any such thing.

> open crate
You open the crate.

In the crate you see a lens.

> take lens
You take the lens from the crate.
```

This is the smallest puzzle there is, and half the puzzles in interactive fiction are elaborations of it. Something is wanted, it is inside something else, and the player has to think of opening it.

A container that is not `openable` has no lid. It is always open, and it cannot be closed.

7.3 How much it holds

The line `holds`, followed by a number, limits how many things a container or a supporter can have in it or on it at once. The crate holds one, and that one is the lens. In a new game, before the lens has been taken out:

```
> open crate
You open the crate.

In the crate you see a lens.

> put taper in crate
There's no room in the crate.
```

`holds` goes with the short facts, on a line of its own. It counts things and knows nothing of their size, so a crate that `holds 1` will take a lens the width of a cartwheel or a wax taper, but not both. Without a `holds` line there is no limit.

The Great Lantern on the roof is now a container as well. It has a seat for exactly one lens, so it `holds 1`, and it has no lid.

```
create the Great Lantern
  aka lantern
  a container, scenery
  in the Lantern Gallery
  holds 1
```

Carry the lens up to the roof and the story lets you seat it, though nothing comes of that until Part III.

```
> put lens in lantern
You put the lens in the Great Lantern.
```

7.4 A cupboard on the wall

The oak press in the Lodge is the same construction as the crate, with `scenery` added so that it stays on its wall.

```
create the oak press
  aka press, cupboard
  a container, scenery, openable, starts closed
  in the Lodge
  holds 6

  A shallow cupboard on the wall with a row of hooks inside, each
  ↪ hook
  under a painted label. Most of the labels have faded past
  ↪ reading.
```

It is empty, and the story notices:

```
> open press
You open the oak press, which is empty.
```

The next chapter hangs a key in it. Since the press is scenery, the room will not mention it, so the Lodge's description gains a clause that does: "and an oak press hangs on the wall beside it".

7.5 Seeing through, and saying it yourself

Two trait words change what the player is told about a holder's contents.

`transparent` is for a container that can be seen into while it is shut. Its contents are visible and cannot be reached. Here is a bell jar from another story:

```
create the bell jar
  aka jar
  a container, openable, transparent, starts closed
  in the Den
  holds 1

  A glass bell jar.

create the moth
  a thing
  in the bell jar

  A grey moth.
```

```
> examine jar
A glass bell jar.

In the bell jar you see a moth.

> take moth
You can't reach the moth.
```

`unlisted` is for a holder whose contents your own prose has already described. Suppose a room's description says that a table stands under the window with a cold supper laid on it. The supper is a thing on the table, and without help the story would add "On the table you see a supper", which says the same thing twice and less well. Mark the table `unlisted`, and that sentence is never printed. The supper is still there to be taken.

```
create the table
  a supporter, scenery, unlisted
  in the Den

  A table laid for supper.
```

7.6 Getting into things

The trait word `enterable` lets the player get into a container or onto a supporter. Hobb's chair becomes a supporter so that things can be put on it, and enterable so that Ansel can sit in it.

```
create the chair
  a supporter, scenery, enterable
  in the Lodge

  A wing chair that has taken the shape of one man.
```

```
> enter chair
You get into the chair.

> north
You're not in a place where you can go anywhere.

> exit
You get out of the chair.
```

The command that works is `enter chair`. A player's first thought will be `sit on chair`, which the story does not yet understand, and chapter 25 adds it in two lines. While Ansel is in the chair the directions do not work, as you would hope.

7.7 Build and try it

Copy `relighting.story` from the companion folder `chapter-07` over your own file. It is the story with this chapter's changes made, and it holds the lines the chapter did not print in full. To make the changes by hand instead, work from this list and take the rest from that file: add the press, the desk, the grimoire, the crate and the lens, change the chair and the Great Lantern to their new kinds, and add the clause about the press to the Lodge's description. Check it and play it.

```
sharpee compose relighting.story --check
sharpee play .
```

Try putting a thing you are not holding onto a supporter. The story picks it up for you first and says that it has.

```
> put kettle on chair
(first taking the kettle)

Taken.

You put the kettle on the chair.

> look
The Porter's Lodge
The porter's lodge is one small room built into the thickness of
the gate arch. A chair, a kettle and thirty years of newspapers
have filled most of it. On the sill a lamp is burning with a
steady blue flame, and an oak press hangs on the wall beside it.
The door to the court is north.

You can see a hand lamp and Hobb here.

On the chair you see a kettle.
```

Then walk to the Undercroft, open the crate, and carry the lens to the roof. That walk is a third of the finished game, and so far nothing stands in its way.

UNDER THE HOOD: ONE TRAIT, TWO ACTIONS · `IF.ACTION.OPENING`

`a container` becomes the container trait and `a supporter` the supporter trait. `openable` adds a third trait, which holds one fact: whether the thing is open. The standard `opening` and `closing` actions change that fact, and they are the only two that do. Every other action reads it. `taking` and `examining` ask whether a container is open before they will look inside, and that single question is the whole of the crate's puzzle. `holds` is the container's capacity, and the `putting` and `inserting` actions read it before they will add anything.

FROM ANOTHER SYSTEM: HOLDERS

Inform 10 has container and supporter as kinds, with openable, transparent and enterable as properties, and a carrying capacity that counts things. Chord's arrangement is the same in every one of those respects, down to the capacity being a count. TADS 3 authors will find it coarser than they are used to. There are no bulk and weight here, and no distinction between a surface, an underside and a rear. A container has one inside, a supporter has one top, and a thing is one or the other.

7.8 What you have so far

Things can now be in and on other things. The lens is in a crate, the grimoire is on a desk, there is a press on the Lodge wall with nothing in it, and the Great Lantern will take a lens. What the story still lacks is anything that stops the player. The next chapter adds the first thing that does: a locked door.

8 Doors, locks and keys

Until now every way in Lanternhall has stood open. This chapter closes one. The Library goes behind a locked door, its key goes into the press in the Lodge, and the player has, for the first time, somewhere they cannot go until they have done something else.

8.1 A door

A door is a kind of its own. It is created like anything else, with one difference you will notice at once.

```
create the library door
  a door, openable, lockable with the library key, starts locked

  Oak, banded with iron, and carved with an open book whose pages
  ↪ have
  been left smooth.
```

There is no placement line. A door is not in a room. It is between two rooms, and it gets its place from the exit that passes through it.

```
create the Cloister, a room
  east to the Refectory
  west to the Library through the library door
  north to the Bell Stair
  down to the Undercroft
```

The words `through the library door`, added to the end of an exit line, put the door on that exit. Because an exit runs both ways, the door stands on both sides. The player can examine it, open it and lock it from the Cloister and from the Library, and you write it once.

If you give a door a placement, or create a door and never put it on an exit, the compiler stops you. This is the second case:

```
relighting.story:86:1 error [analysis.door-unconnected] Door
`library door` is never referenced by a `through` exit line — an
unconnected door is unanswerable (its room pair could never
resolve). Add `<direction> to the <room> through the library door`
on a room.
```

A door begins closed, and the player must open it before walking through. That is true even of a door written with nothing but its kind, `a door`.

8.2 A lock and its key

The library door has three trait words after its kind. `openable` you know from the crate. The other two are new.

`lockable with the library key` gives the door a lock and names the thing that works it. This is the first trait word you have met that takes more words after it. Those words are called a **setting**, and here the setting is the key.

`starts locked` says how the door begins. Like `starts closed`, it is a starting state, and it needs its trait word beside it. Write `starts locked` on something that is not `lockable` and the compiler objects:

```
relighting.story:87:21 error [analysis.starts-state-pairing]
`starts locked` requires `lockable` composed on this entity.
```

The key is an ordinary thing. Nothing in its own block says that it is a key. It is a key because a lock names it.

```
create the library key
  a thing
  in the oak press

  A long key with a book cut into its bow.
```

It hangs in the oak press in the Lodge, which chapter 7 left empty. The press is closed, so the key cannot be seen until the press is opened, and now the story has a chain of three steps: open the press, take the key, unlock the door.

8.3 Playing it

Walk to the Cloister without the key and try the door.

```
> west
The library door is locked.

> open door
The library door is locked.
```

Go back for the key, and try again.

```
> open press
You open the oak press.

In the oak press you see a library key.

> take key
You take the library key from the oak press.
```

```
> unlock door with key
You unlock the library door with the library key.

> west
The library door is closed.

> open door
You open the library door.

> west
Library
The reading room of Lanternhall, with its desks in rows and its
galleries of shelves. The stacks run back to the north, out of the
light. The cloister is east.

On the reading desk you see a grimoire.
```

Every one of those replies is the platform's. You declared a door, a lock and a key, and the refusals, the unlocking and the opening all followed. The same is true of the wrong key, tried while the door is still locked:

```
> unlock door with taper
The wax taper doesn't fit the library door.
```

8.4 Saving the player a step

Unlocking a door and then opening it are two commands for one intention. If you would sooner the door opened as it unlocked, add one line to its block.

```
create the library door
  a door, openable, lockable with the library key, starts locked
  opens when unlocked
```

```
> unlock door with key
You unlock the library door with the library key, and it swings
open.
```

The Relighting leaves this line out, because it wants the player to stand at the open door for a moment before going in. It is a matter of pace, and yours to choose for each door.

8.5 Locks on other things

A lock is not only for doors. Anything `openable` may be `lockable` as well, with the same words. A strongbox is a container that is `openable, lockable with the brass key, starts locked`, and it behaves as the door does: it cannot be opened until it is unlocked, and it is unlocked with its key.

One key can work several locks. Name the same key in each of them.

8.6 Naming keys

Chapter 6 warned about two things that answer to the same word. Keys are where that warning matters most, because stories collect them. In a draft of this chapter Ansel arrived carrying a ring of keys, and with the library key in the same room, `take key` had two possible meanings. The story asked:

```
> take key
Which do you mean: the key ring or the library key?
```

The ring was taken out of the story for this reason alone. You cannot give two keys names that share no word, since both are keys, so plan instead for them never to be carried at once, or accept that the player will sometimes be asked.

8.7 Build and try it

Copy `relighting.story` from the companion folder `chapter-08` over your own file. It is the story with this chapter's changes made. To make

the changes by hand instead, work from this list and take the rest from that file: add the door and the key, put the door on the Cloister's west exit, and change "the library west" to "the library door west" in the Cloister's description.

```
sharpee compose relighting.story --check
sharpee play .
```

Play the chain from the Lodge to the Library. Then, standing in the Library, close the door and lock it behind you, and try to leave.

```
> close door
You close the library door.

> lock door with key
You lock the library door with the library key.

> east
The library door is locked.
```

The door is the same door from both sides.

UNDER THE HOOD: ONE FACT, MANY READERS · `IF.ACTION.UNLOCKING`

`lockable` becomes the lockable trait, which holds two facts: whether the thing is locked, and which thing is its key. The standard `locking` and `unlocking` actions change the first fact, and they check the second before they do. The `opening` action reads the lock and refuses a locked thing, and the `going` action reads the door on an exit and refuses a closed one. That is why you got two different refusals at the same door, "locked" and then "closed": two actions were each reading the fact that concerned them.

FROM ANOTHER SYSTEM: DOORS

In Inform 10 a door is a kind: you connect two rooms by way of a named thing, say that the thing is a door, and give a lock its matching key. In TADS 3 a door is usually two objects, one for each side, linked to each other. Chord's door is one thing with no location of its own, attached to an exit. You never make a second side, and you never say where the door is. If your instinct is to place it in a room, the compiler will remind you not to.

8.8 What you have so far

The Library is locked, and the way in runs through the Lodge. The Relighting has its first puzzle with more than one step. Everything in it so far, though, is a thing to be carried, opened or unlocked. The next chapter surveys the other things a thing can be.

9 What things can do: the trait words

You have met a handful of trait words already: `scenery`, `plural`, `proper`, `concealed`, `openable`, `lockable`, `transparent`, `unlisted` and `enterable`. Each one is an adjective after the kind that gives a thing an ability or a manner. This chapter is a tour of the ones that let the player do something to a thing: wear it, read it, eat it, switch it on, pull it, push it and break it.

The pattern is the same every time, and it is worth stating once. A trait word pairs a thing with one of the platform's standard actions. The action already exists, the player already knows its verbs, and all of its replies are already written. Your one word says that this thing is the kind of thing the action applies to. Without the word, the action refuses politely. With it, the action works.

The list of trait words is closed. You choose from it and cannot add to it by typing a new adjective. Chapter 24 shows the proper way to make a trait of your own.

9.1 Wearing

`wearable` lets a thing be put on and taken off.

```
create the travelling coat
  aka coat
  a thing, wearable

  A long coat the colour of the road.
```

A person can begin the story wearing something. The line is `wears`, and it sits beside `carries` in the person's block.

```
create Ansel
  a person, proper
  playable
  starts in the Lodge
  carries the wax taper
  wears the travelling coat
```

```
> inventory
You are carrying and wearing:

a wax taper

a travelling coat (worn)

> take off coat
You take off the travelling coat.

> wear coat
You put on the travelling coat.
```

A bare `wearable` thing can be worn with anything else. If two things ought to compete for the same place, say where each is worn, with `wearable on the` followed by a word of your own for the part of the body. The Relighting has one coat and nothing to compete with it, so here are a hat and a cap from another story, which has a room called the Den:

```
create the hat
  a thing, wearable on the head
  in the Den

  A hat.

create the cap
  a thing, wearable on the head
  in the Den

  A cap.
```

```
> wear hat
(first taking the hat)

Taken.

You put on the hat.

> wear cap
You're already wearing the hat on your head.
```

9.2 Reading

`readable` lets a thing be read, and it is the first trait word in this chapter that needs a setting. The setting is the text, written `with text` and then the words in quotation marks.

```
create the grimoire
  aka book, margin, note
  a thing, readable with text "In the margin, in a small brown
  ↪ hand: gnusto. Came over the water as a working, they say.
  ↪ Meaning lost."
  on the reading desk

  A book as long as your forearm, chained to nothing any more. Its
  ↪ pages
  are blank, or were. Here and there a line of ink has come back,
  ↪ as
  frost comes onto a window. Someone has written in one margin.
```

This is one of the places chapter 1 promised, where quotation marks are needed. The text is one line, however long, and it cannot itself contain a quotation mark.

A readable thing has two pieces of prose, and the player reaches them with two different verbs. The description is what the thing looks like. The text is what it says.

```
> examine book
A book as long as your forearm, chained to nothing any more. Its
pages are blank, or were. Here and there a line of ink has come
back, as frost comes onto a window. Someone has written in one
margin.

> read book
(first taking the grimoire)

You take the grimoire from the reading desk.

The grimoire reads:
In the margin, in a small brown hand: gnusto. Came over the water
as a working, they say. Meaning lost.
```

The description ends by pointing at the margin, because a player will not think to read a blank book unless something tells them to. The same goes for the `aka` line: once the description has said "margin", `read margin` has to work.

Always give `readable` its text. The compiler accepts the word alone, and the player is then shown "The notice reads:" with nothing after it.

That word in the margin, by the way, is the only piece of old lore in The Relighting. The people of this country say *gnusto* of sunlight, as a blessing and a habit of speech, and the scholars of the Hall always held that it came from somewhere else and once meant something more.

9.3 Eating and drinking

`edible` lets a thing be eaten, and **`drinkable`** lets it be drunk. Once consumed, the thing is gone from the story.

```
create the seed cake
  aka cake
  a thing, edible
  on the chair
  servings 2

  Half a seed cake on a plate, cut with some care into two pieces.
```

The line `servings`, with a number, says how many times the thing can be eaten before it is finished. Without it, one bite is the whole thing.

```
> eat cake
(first taking the seed cake)

You take the seed cake from the chair.

You eat some of the seed cake.

> eat cake
You eat the seed cake.
```

A drinkable thing is written the same way, and `servings` counts its sips.

9.4 Pulling and pushing

`pullable` and **`pushable`** let a thing answer `pull` and `push`. By themselves they do very little: the action succeeds, the player is told so, and nothing in the world changes.

```
create the bell rope
  aka rope
  a thing, scenery, pullable
  in the Bell Stair

  A rope as thick as your wrist, coming down out of the dark.
```

```
> pull rope
You pull the bell rope.

> take rope
The bell rope is fixed in place.
```

That is deliberate. Pulling a rope ought to ring a bell, and what a pull causes is for you to say. Part III teaches how, and the rope is here now so that there is something to attach the consequence to. A lever, a cord, a handle and a drawer are all pullable things waiting for chapter 14.

A pushable thing has one ability that needs no further writing. It can be pushed from one room to another, if you say between which rooms. Nothing in The Relighting travels this way, so here is a trolley from the same other story, whose Den has a Yard to the north of it:

```
create the trolley
  a thing, pushable
  in the Den

  A trolley.

  pushing
    between the Den and the Yard
  end pushing
```

The `pushing` block lists pairs of rooms. A row beginning `between` allows the push both ways, and a row written `from the Den to the Yard` allows it one way. There must be an ordinary exit between the two rooms for the trolley to travel along.

```
> push trolley north
You push the trolley north.

Yard
The yard.

You can see a trolley here.
```

9.5 Breaking

`breakable` lets a thing be broken by attacking it. Add **`fragile`** and it also breaks when it is dropped.

```
create the cup
  aka teacup
  a thing, breakable, fragile
  on the chair

  A china cup without a saucer, with a brown ring where the tea
  ↪ stood.
```

```
> take cup
You take the cup from the chair.

> drop cup
The cup hits the floor and breaks.
```

A broken thing is still there and can still be examined. Its description does not change by itself, which is another matter for Part III. A thing may also begin the story broken, with `starts broken` beside `breakable`.

9.6 Fixed to something

`attached to the`, followed by the name of another thing, fixes one thing to another. An attached thing cannot be taken. Add **`detachable`** and a pull frees it. The Relighting fastens nothing in this way, so the tassel is from the other story too, and it hangs from a hook in the Den:

```
create the tassel
  a thing, attached to the hook, detachable
  in the Den

  A tassel.
```

```
> take tassel
The tassel is fixed to the hook.

> pull tassel
You pull the tassel free of the hook.

> take tassel
Taken.
```

This differs from `scenery` in what the player is told and in what can change. Scenery is fixed in place for good. An attached thing is fixed to a particular other thing, the refusal names it, and the attachment can end.

9.7 Switching on and off

`switchable` gives a thing an on and an off. The wireless is the last thing from the other story:

```
create the wireless
  a thing, switchable
  in the Den

  A wireless set.
```

```
> switch on wireless
The wireless hums to life.

> switch off wireless
The wireless powers down with a soft whir.
```

Like pulling, switching changes one fact about the thing and leaves the consequences to you. A switchable thing begins off unless you write `starts on`. Lanternhall has stood dark for thirty years and has nothing in it to switch on, so The Relighting does not use this word.

9.8 The ones that wait

Five more trait words are on the list, and each depends on something this book has not reached. They are named here so that you know they exist.

- `cuttable`, `diggable` and `flammable` let a thing be cut, dug or burned with a named tool. Each needs you to say what the cutting, digging or burning does, which is a clause from chapter 14.
- `settable` makes a thing a dial with named positions. The positions are states, from chapter 13.
- `costs` puts a price on a thing. The price is paid from a counter, from chapter 18.

The trait words that concern light and sound have the next chapter to themselves, and Appendix B lists every trait word with its settings.

9.9 Build and try it

Copy `relighting.story` from the companion folder `chapter-09` over your own file. It is the story with this chapter's changes made, and it holds the lines the chapter did not print in full. To make the changes by hand instead, work from this list and take the rest from that file: add the coat, the seed cake, the cup, the candles and the bell rope, make the grimoire readable, give Ansel the `wears` line, and mention the candles in the Refectory's description.

```
sharpee compose relighting.story --check
sharpee play .
```

Then do to each new thing what it is for, and also what it is not for. The refusals are as much a part of the story as the successes, and you should know what your player will read.

```
> read coat
There's nothing written on the travelling coat.

> eat coat
That's not something you can eat.
```

UNDER THE HOOD: A TRAIT AND ITS ACTION · SHARPEE COMPOSE

Each trait word in this chapter adds one trait to the thing, and each trait has a standard action that looks for it. `wearable` is read by `wearing` and `taking_off`, `edible` by `eating` and `drinking`, `readable` by `reading`, `switchable` by `switching_on` and `switching_off`, `pullable` by `pulling`, and `pushable` by `pushing`. The action's first step is to ask whether the thing has the trait, and "That's not something you can eat" is the answer when it does not. A setting, such as the text of a readable thing or the servings of an edible one, is a field on the trait.

FROM ANOTHER SYSTEM: PROPERTIES AND CLASSES

An Inform 10 author would write most of this chapter as properties: a thing is wearable, edible, pushable between rooms, or switched on. A TADS 3 author would reach for classes such as a wearable, a food or a switch. Chord's trait words are closer to the Inform 10 properties in how they read and closer to the TADS 3 classes in what they are, since each one brings behaviour with it and not only a flag. What neither system prepares you for is that the list is closed. In Chord you do not add a property to a thing by declaring it. You compose a thing from the trait words that exist, and you define a new trait, with its own behaviour, when none of them fits.

9.10 What you have so far

Ansel arrives in a coat, Hobb's cake can be eaten and his cup broken, the grimoire can be read, and the bell rope can be pulled to no effect. The next chapter turns to the trait words the story is named for: the ones that give light.

10 Light, dark and the other senses

The Relighting is a story about a light that has gone out, and so far every room in it is as bright as every other. This chapter makes one room dark, makes the hand lamp a lamp, and then looks briefly at what the player can smell and hear.

10.1 A dark room

A room is lit unless you say otherwise. The word that says otherwise is `dark`, and it goes on the room's header line after the kind.

```
create the Undercroft, a room, dark

  Low vaults on squat pillars, running off under the whole of the
  ↪ hall.
  The air is cold and smells of stone. The steps go back up.
```

`dark` is a trait word for rooms only. In a dark room the player cannot see. The room's description is not shown, the things in it are not listed, and nothing there can be examined or taken, because as far as the player can tell nothing is there.

```
> down
It is pitch dark. You are likely to be eaten by a grue.

> look
It's pitch dark, and you can't see a thing.

> examine crate
You can't see any such thing.

> up
Cloister
A covered walk around a garth gone to nettles. The refectory is
east, the library door west, and the foot of the bell stair north.
Worn steps go down into the dark under the hall.
```

Notice three things. The player was allowed to walk into the dark, and was allowed to walk out of it again. The exits of a dark room still work, so a player who blunders in is never trapped. And the first line the player read came from another story entirely.

The grue is the platform's stock line for entering a dark room, kept in honour of the games that invented this kind of puzzle. It has no place in Lanternhall, where the only thing in the dark is a crate. Chapter 22 shows how to replace any of the platform's messages with your own, and this is the first one The Relighting replaces. Until then, leave it. Knowing which lines in your story are not yours is worth a little embarrassment.

10.2 A light to carry

A thing gives light if it is a `light-source`, and you say how it begins.

```
create the hand lamp
  a thing, light-source, starts unlit
  in the Lodge

  A tin hand lamp with a wire handle. Its wick is dry and its
  ↪ glass is
  clean, which is Hobb all over.
```

`starts lit` and `starts unlit` are the two starting states, and like every starting state they need their trait word. A thing that is not a `light-source` cannot start lit:

```
relighting.story:259:12 error [analysis.starts-state-pairing]
`starts lit` requires `light-source` composed on this entity.
```

A light source that is lit lights the room it is in. It does so whether the player is holding it or has put it down, and it goes on doing so if the player leaves and comes back.

```
> light lamp
You light the hand lamp.
```

```
> down
Undercroft
Low vaults on squat pillars, running off under the whole of the
hall. The air is cold and smells of stone. The steps go back up.

You can see a crate here.
```

The player lights and puts out a light source with the ordinary verbs, and the platform notices when doing so changes what can be seen.

```
> extinguish lamp
You put out the hand lamp, and darkness closes in.

> look
It's pitch dark, and you can't see a thing.

> light lamp
You light the hand lamp, and the darkness draws back.

Undercroft
Low vaults on squat pillars, running off under the whole of the
hall. The air is cold and smells of stone. The steps go back up.

You can see a crate here.
```

Only a light source can be lit. Ansel's taper is a plain thing, and the story says so:

```
> light taper
The wax taper isn't something you can light.
```

10.3 What this does to the story

With two words added, The Relighting has changed shape. The lens is in a crate in the Undercroft, the Undercroft is dark, and the lamp is in the Lodge. A player who goes straight down the steps learns that they need a light, and has to go back for one. The world now has an order in which things must be done, and you wrote no rule to enforce it. The darkness enforces it.

There is one thing wrong, and it is worth seeing clearly. The lamp lights when the player types `light lamp`. In the finished story, lighting that lamp is the first act of magic anyone has performed in thirty years, and it should not be available for the asking. For now it is. The standard action that lights a light source does not know that this one is special. Teaching it takes a requirement from chapter 15 and a new verb from chapter 25, and until then the lamp is merely a lamp.

10.4 Seeing in the dark

A person may be given **`night-vision`**, which lets them see in a dark room as though it were lit. It is a trait word for people only.

```
create Ansel
  a person, proper, night-vision
  playable
  starts in the Cellar
```

For a person who is the player, the dark simply stops mattering. Ansel has no such gift. The word is more often useful on someone else, in a

story where a character must find their way about a dark house on their own turns, as Part V describes.

10.5 Smell and sound

The player has other senses, and two trait words speak to them.

A thing that is **`smelly`** can be smelled from anywhere in its room, and a thing that is **`loud`** can be heard. What makes these two worth having is that they pass through a closed container. Here is a tin with three things in it, from another story.

```
create the tin
  a container, openable, starts closed
  in the Den

  A tin.

create the cheese
  a thing, smelly
  in the tin

  A ripe cheese.

create the clock
  a thing, loud
  in the tin

  A clock.

create the sock
  a thing
  in the tin

  A sock.
```

With the tin shut, the player can see none of the three, and can detect two.

```
> smell
You smell a cheese.

> listen
You hear a clock.

> listen to tin
You hear faint sounds from inside the tin.
```

This is a way of telling the player that a closed thing is worth opening without saying so. The sock, which is neither smelly nor loud, gives no sign of itself until the tin is open.

Sound goes much further than this. A story can define sounds of its own, play them, and have people in the story react to hearing them, and The Relighting does all three with its tower bell. Those need clauses from Part III and are taught in chapter 36.

10.6 Build and try it

Copy `relighting.story` from the companion folder `chapter-10` over your own file. It is the story with this chapter's changes made. To make the changes by hand instead, work from this list and take the rest from that file: add `dark` to the Undercroft, make the hand lamp a light source, and let the Cloister's description warn of the dark at the bottom of the steps.

```
sharpee compose relighting.story --check
sharpee play .
```

Walk down without the lamp, then fetch it, light it and walk down with it. Then try leaving the lit lamp in the Undercroft and coming back.

```
> drop lamp
Dropped.

> up
Cloister
A covered walk around a garth gone to nettles. The refectory is
east, the library door west, and the foot of the bell stair north.
Worn steps go down into the dark under the hall.

> down
Undercroft
Low vaults on squat pillars, running off under the whole of the
hall. The air is cold and smells of stone. The steps go back up.

You can see a crate and a hand lamp here.
```

UNDER THE HOOD: PERCEPTION · `IF.EVENT.PERCEPTION.BLOCKED`

Darkness is not handled by the actions. The `looking` action describes the Undercroft in exactly the same way whether or not there is light. Between the action and the player sits a perception step, which asks whether the player can see where they are. If they cannot, it replaces what the action reported with a single event saying that sight was blocked, and "It's pitch dark, and you can't see a thing" is how that event is worded. This is why you did not have to write a dark version of the room. There is only one version, and the platform decides whether the player gets it.

FROM ANOTHER SYSTEM: LIGHT

Both systems have this model, and it will feel like home. Inform 10 has dark rooms and lit things, and TADS 3 has a brightness on rooms and on light sources. Chord's version is the simple one: a room is dark or it is not, a light source is lit or it is not, and one lit source in the room is enough. There are no degrees of brightness. The surprise for both kinds of author is further on, in that lighting a light source is an ordinary standard action that you restrict, not a rule you write from nothing.

10.7 What you have so far

Lanternhall has a dark place under it and one lamp to see by, and the story has its first forced order of events. The next chapter steps back from single rooms and things to look at the map as a whole.

11 Regions

Rooms belong together. The Lodge and the Gate Court are the part of Lanternhall a visitor is allowed to see, and everything past the cloister arch is the college proper. A **region** is how a story says that. It is a named group of rooms, and once the group has a name, other parts of the story can speak about all its rooms at once.

A region changes nothing the player can do. It is a tool for you. This chapter uses it for three things: placing scenery across many rooms, making a run of similar rooms without writing each one, and laying out a set of exits as a table. Part III adds a fourth, which is making something happen when the player crosses from one region to another.

11.1 Making a region

A region is created like anything else, and it is the last of the kinds.

```
create the Grounds, a region

create the Hall, a region
```

A region has no description, because the player is never in a region. They are in one of its rooms. It has no placement either, and a region cannot be put inside another one.

11.2 Putting rooms in it

A room joins a region with a line in its own block, `in the` followed by the region's name.

```
create the Lodge, a room
  in the Grounds
  north to the Gate Court

create the Gate Court, a room
  in the Grounds
  exits: south to the Lodge, north to the Cloister
```

The six rooms behind the arch each take `in the Hall` in the same way. A room may be in one region or in none. Two `in` lines on one room is an error:

```
relighting.story:35:3 error [analysis.room-two-regions] `Lodge`
already names a region on line 34 — a room is in at most one
region. Remove one of the `in` lines.
```

This is the same word `in` that places a thing in a room, and the compiler tells the two apart by what is being placed and where. A room goes in a region. A thing goes in a room, and a thing may not be placed in a region directly, since it would then have no particular room to be in. There is one exception, and it is the useful one.

11.3 Scenery for a whole region

Chapter 6 made the gate a backdrop by listing its rooms. Scenery placed in a region is a backdrop in every room of that region, with no list to keep up to date.

```
create the bell tower
  aka tower
  a thing, scenery
  in the Grounds

  The tower of Lanternhall stands over everything, square and
  ↪ grey, with
  the dark glass of the Great Lantern at the top of it.
```

The tower can be examined from the Lodge and from the Gate Court. Past the arch the player is underneath it and cannot see it.

```
> examine tower
The tower of Lanternhall stands over everything, square and grey,
with the dark glass of the Great Lantern at the top of it.

> north
Cloister
A covered walk around a garth gone to nettles. The refectory is
east, the library door west, and the foot of the bell stair north.
Worn steps go down into the dark under the hall.

> examine tower
You can't see any such thing.
```

If a room is later added to the Grounds, the tower is visible from it without another word from you. That is the argument for regions in one example: say a thing once, at the level where it is true.

11.4 A run of rooms

The Library's description says that its stacks run back to the north. They are three rooms, alike in everything but their number. Writing three `create` blocks with the same description would be tedious and would invite the three to drift apart, so a region can make them as a group.

```
create the Hall, a region
  rooms Stack 1 to 3

    Shelves rise on both sides to a ceiling you cannot make out.
    ↪ Every
    spine is blank.
```

The line `rooms Stack 1 to 3` makes three rooms, named `the Stack 1`, `the Stack 2` and `the Stack 3`, all of them in the Hall. The paragraph beneath it, after a blank line and indented one level further, is the description they share. A group may also carry trait words

such as `dark`, a `first time` block and a `room name` block, and each room in the group gets its own copy.

The rooms are real rooms. You can name any of them elsewhere in the file, place things in them, and give one of them an exit the others do not have.

11.5 A table of exits

The three stacks need joining to each other and to the Library. You could write those exits in the Library's block, but the stacks have no blocks of their own to hold the rest. A region can instead carry a table of exits, with one row for each room that has any.

```
exits
  the Stack 1: south to the Library, north to the Stack 2
  the Stack 2: north to the Stack 3
```

The table begins with the word `exits` on a line by itself, with no colon. Each row is a room of the region, a colon, and that room's exits separated by commas. The exits are the same as exits written anywhere else. They work both ways, so these two rows make the whole run from the Library to the last stack and back.

```
> north
Stack 1
Shelves rise on both sides to a ceiling you cannot make out. Every
spine is blank.

> north
Stack 2
Shelves rise on both sides to a ceiling you cannot make out. Every
spine is blank.

> north
Stack 3
Shelves rise on both sides to a ceiling you cannot make out. Every
spine is blank.

> north
You can't go that way.
```

Every row of the table must be a room of that region. The Library is in the Hall, so `south to the Library` is allowed as a destination, and a row for a room of the Grounds would be refused.

Be careful not to confuse the table with the `exits:` line from chapter 5. The line has a colon after `exits` and lives in a room's block. The table has no colon after `exits`, lives in a region's block, and has its colons after the room names.

11.6 A backdrop for the stacks

The stacks, like every room, have made a promise: shelves and spines. A backdrop over the Library and its three stacks keeps it.

```
create the shelves
  aka shelf, spines, books
  a thing, scenery, plural
  in the Library, the Stack 1, the Stack 2 and the Stack 3

  Thousands of books, shelved by a system nobody alive remembers.
  ↪ Take
  one down and its pages are as blank as its spine.
```

These four rooms are not a region of their own, and the Hall is too large, since the shelves should not follow the player into the Refectory. So this backdrop names its rooms. Use a region when the scenery belongs to all of it, and a list when it does not.

11.7 Build and try it

Copy `relighting.story` from the companion folder `chapter-11` over your own file. It is the story with this chapter's changes made. To make the changes by hand instead, work from this list and take the rest from that file: add the two regions, put each room in one, and add the stacks, the tower and the shelves. This book places the regions at the top of the file, after the header, so that the plan of the place is the first thing you read.

```
sharpee compose relighting.story --check
sharpee play .
```

Walk the stacks to the end and back, and examine the tower from each side of the arch. Then try placing the kettle in the Grounds instead of the Lodge, and read what the compiler says.

```
relighting.story:142:3 error [analysis.thing-in-region] `kettle`
is placed in the region `Grounds` — a region holds rooms, not
objects. Which of its rooms is `kettle` in? Name that room
instead.
compose: relighting.story failed the load-time gates (1 error(s))
```

Put the kettle back in the Lodge, and check that the story is gate-clean again.

UNDER THE HOOD: A REGION IS AN ENTITY TOO · SHARPEE COMPOSE

In the Story IR a region is an entry in `entities` like a room or a thing, with the kind `region`. A room's `in the Hall` is recorded as the room being contained in that entity, and a group such as `rooms Stack 1 to 3` is expanded by the compiler into three ordinary rooms before the platform ever sees it. The table of exits is expanded in the same way, into exits on the rooms themselves. By the time the story runs, nothing distinguishes a stack from a room you wrote out by hand.

FROM ANOTHER SYSTEM: REGIONS

Inform 10 has regions, and they are used as they are here: rooms are declared to be in one, a backdrop can be placed in one, and rules can be written that affect one region at a time. The differences are that Inform 10's regions can be put inside each other and Chord's cannot, and that Chord adds the room group and the exit table. TADS 3 has no region class in its standard library. If that is your background, the region is the thing you used to build by hand.

11.8 What you have so far

Lanternhall is two regions and eleven rooms, with a tower over the Grounds and blank books by the thousand in the Library. Every way in it but one stands open. The last chapter of Part II shows the other things a way can be: blocked, deadly, and hidden behind a hearth.

12 Ways that are closed, hidden or strange

An exit, so far, is either there or not there, and the only thing that has ever stood in one is a door. This chapter adds four more possibilities. A way can refuse the player with words of your choosing. It can kill them. It can exist only some of the time. And it can be a way that no compass direction describes, taken by typing a command of its own.

12.1 A way that refuses

The stair to the roof of Lanternhall has been sealed for thirty years by an iron grille that nobody built. The exit is there, and the player may not use it.

```
create the Bell Stair, a room
  in the Hall
  up to the Lantern Gallery
  up is blocked: stair-sealed

  The foot of the tower. A stair winds up around the well where
  ↪ the bell
  rope hangs, and an iron grille stands across its third step. The
  cloister is south.
```

The new line reads as English does: `up is blocked`, a colon, and a name. The name, `stair-sealed`, is a **phrase key**. It stands for a piece of text that is written elsewhere in the file, in a block near the end.

```
define phrases en-US
  stair-sealed:
    The grille stands across the stair, and there is no way round
    ↪ it.
```

This is the first phrase in The Relighting, and phrases have a whole chapter to themselves, which is chapter 20. What you need now is small. A `define phrases` block is a list of keys, each followed by a colon and then its text on the lines beneath, indented one level further. The words `en-US` say which language the text is in. A key is lower-case words joined by hyphens, and you make it up.

When the player tries the stair, the phrase is what they read.

```
> up
The grille stands across the stair, and there is no way round it.
```

You cannot write the text on the `blocked` line itself. What follows the colon is always a key. If the key has no phrase, the compiler says which one is missing:

```
relighting.story:102:3 error [analysis.missing-phrase] Phrase
`stair-sealed` is not declared in en-US.
```

A blocked way wants something in the room to account for it, so the grille is a thing as well.

```
create the iron grille
  aka grille
  a thing, scenery
  in the Bell Stair

  Wrought iron, floor to vault, with no lock, no latch and no
  ↪ hinges.
  It was not built. It was put there.
```

The grille and the blocked line have nothing to do with each other as far as the story is concerned. The line stops the player, and the thing gives them something to look at. Keeping them in step is your job.

As it stands the stair is blocked for good, which makes the Lantern Gallery unreachable. A blocked line may carry a condition, written `up is`

`blocked while` and then the condition before the colon, and then the way is blocked only while the condition holds. Lifting the grille is the business of chapter 24. Conditions are chapter 16.

12.2 A way that kills

On the roof, a low parapet runs along the east side of the Gallery. There is no room to the east. There is a drop.

```
create the Lantern Gallery, a room
  in the Hall
  east is deadly: over-the-parapet
```

```
define phrases en-US
  over-the-parapet:
    You step up onto the parapet to see the court. The court comes
    ↪ up to
    meet you.
```

A deadly way needs no exit line, because it leads nowhere. When the player takes it, the phrase is printed and the story ends.

```
> east
You step up onto the parapet to see the court. The court comes up
to meet you.

[The story has ended. RESTART, RESTORE or UNDO]
```

Be sparing with this, and be fair. The Gallery's description now says, "To the east a low parapet is all there is between you and the court." A player who has read that and walks east has been warned. A deadly way that the prose did not signal is a trap, and players forgive very little less.

Because the stair is sealed, you cannot reach the parapet in the story as it stands. To see it work, wait until this chapter's story is in your folder, which is the copy in "Build and try it" below. Then take out the `up is blocked` line for a moment, walk up and step off, and put the line back.

12.3 A way that is sometimes there

A blocked way is one the player can see and cannot take. A hidden way is not there at all until something makes it so. An exit line may end with `while` and a condition, and the exit then exists only while the condition is true.

```
create the Study, a room
  north to the Vault while the bookcase is open
```

Until the bookcase is open, going north from the Study gets the same answer as any direction with no exit. Nothing tells the player that a way exists. When the bookcase opens, the exit is simply there.

The difference between the two is what the player knows. Use `blocked` when they should see the obstacle and wonder how to get past it. Use a `while` exit when discovering that the way exists is itself the puzzle.

The Relighting has no hidden exit of this kind. It has something stranger.

12.4 A way with its own command

Behind the Refectory's hearth, the iron plate does not sit square, and there is a gap a thin person could get through. It leads down into the Undercroft. It is not north or east or down. It is behind the hearth, and the player gets through it by saying so.

```
create the Refectory, a room
  in the Hall

  Three long tables, each with its row of candles, and a hearth
  ↪ you
  could stable a horse in. The benches are stacked against the
  ↪ wall,
  where someone left them for the holidays. The cloister is back
  ↪ to the
  west.

  passage to the Undercroft
    grammar
      squeeze or crawl behind [the] hearth
  end passage
```

A **passage** is an exit that is taken with a command instead of a direction. The block opens with `passage to` and the room it leads to, and it closes with `end passage`. Inside it, the word `grammar` stands on a line by itself, and the lines beneath that are the commands, indented one level further.

The command line is a pattern, and it has two small devices in it.

- `squeeze or crawl` means that either word will do.
- `[the]` means that the word in brackets may be typed or left out.

So this one line accepts `squeeze behind hearth`, `crawl behind hearth`, `squeeze behind the hearth` and `crawl behind the hearth`. Chapter 23 uses the same notation at length, for verbs of your own.

```
> examine hearth
The back of the hearth is a single iron plate, and it does not sit
quite square. There is a hand's breadth of darkness down one side.

> squeeze behind hearth
It is pitch dark. You are likely to be eaten by a grue.
```

The Undercroft is dark, and Ansel has arrived in it without the lamp. That is the passage working as intended, and with a lit lamp in hand the same command shows the room.

Three properties of a passage are worth knowing.

A passage is one-way. The gap behind the hearth goes from the Refectory to the Undercroft, and there is no climbing back through it. If you want a way back, write a second passage in the other room.

Its commands are understood only in its own room. Type `squeeze behind hearth` in the Undercroft, where there is no such passage, and the story does not know the words:

```
> squeeze behind hearth
I don't understand that.
```

And nothing announces it. The player has to find the passage by reading. Here the hearth's description does the work: "a hand's breadth of darkness down one side" is an invitation, and `squeeze` is the verb it invites.

A passage may also be blocked or deadly, with a line inside the block: `blocked: stair-sealed`, or `deadly: over-the-parapet`, each naming a phrase key as before.

12.5 Build and try it

Copy `relighting.story` from the companion folder `chapter-12` over your own file. It is the story with this chapter's changes made. To make the changes by hand instead, work from this list and take the rest from that file: add the two lines to the Bell Stair and the Lantern Gallery, the passage to the Refectory, the grille, and the `define phrases` block. The phrases go after the last `create` block and before the start block. Nothing is raised in the header: this is still version `0.2.0`, and it is the end of Part II.

```
sharpee compose relighting.story --check
sharpee play .
```

Fetch the lamp and light it, then go through the hearth.

```
> squeeze behind hearth
Undercroft
Low vaults on squat pillars, running off under the whole of the
hall. The air is cold and smells of stone. The steps go back up.

You can see a crate here.
```

UNDER THE HOOD: REASONS TO REFUSE · `IF.ACTION.GOING`

All four of these are the same standard action, `going`, being given something to consult. A `blocked` line is a refusal with a message attached. A `deadly` line is a refusal that also ends the story. An exit with a `while` condition is an exit the action cannot find until the condition holds. A passage adds a command to the story's grammar that leads to `going`, but only in the room that has the passage. In every case you have described the way, and the one action that moves the player has decided what to do about it.

FROM ANOTHER SYSTEM: UNUSUAL EXITS

In Inform 10 you would write a rule that stops the going action and prints a reason, and you would write a new action, with its own grammar line, for squeezing behind the hearth. In TADS 3 the direction property of a room can hold a message or a special connector object in place of a destination. Chord treats each of these as a fact about the way and gives each its own line. The gain is that the map remains readable as a map: every way out of a room, open or closed or strange, is in that room's block. The cost is that you have these forms and no others, until Part III gives you clauses that can refuse anything at all.

12.6 What you have so far

That is the world. Lanternhall has eleven rooms in two regions, a locked door, a dark cellar, a sealed stair, a hidden way and a deadly one. It has a porter, a coat, a cake, a cup, a lamp, a lens in a crate and a book with one word in its margin. A player can walk it, pick it up, open it, wear it and unlock it.

What a player cannot do is change it. Nothing in Lanternhall yet reacts to anything. The rope rings no bell, the lens seats in the Lantern to no effect, and the lamp lights for anyone who asks. Part III is about making things happen.

PART III. MAKING THINGS HAPPEN

13 States

Part II built a world that a player can walk through and pick up. Nothing in it remembers anything. The grille is across the stair because a line says so, and the lens is cracked because its description says so, and neither of those is a fact the story could act on.

A **state** is such a fact. It is a word that says what condition a thing is in, chosen from a short list that you write. This chapter gives states to the story and to five of its things. The next chapter makes them change.

13.1 Declaring states

The iron grille across the Bell Stair is either standing or it has been lifted. That is one line in its block.

```
create the iron grille
  aka grille
  a thing, scenery
  in the Bell Stair
  states: standing, lifted

  Wrought iron, floor to vault, with no lock, no latch and no
  ↪ hinges.
  It was not built. It was put there.
```

The line is the word `states`, a colon, and the names separated by commas. A state name is one word, or several joined by hyphens. Three rules come with the line.

- A thing is always in exactly one of its states.
- The first state listed is the one it starts in. The grille starts `standing`.
- The list has a direction. A thing may move to a later state and may not come back to an earlier one, unless you say otherwise.

The last rule is the one that surprises people, and it is deliberate. Most things in a story happen once. A seal is broken, a page is read, a bridge falls. Writing the states in order tells the compiler which way the story runs, and it will stop you if a later line tries to run it backwards.

Here are the other things in The Relighting that get states in this chapter.

```
create the lens
  a thing
  in the crate
  states: cracked, sound
```

```
create the cistern
  aka ice
  a thing, scenery
  in the Undercroft
  states: frozen, thawing, running

  A stone cistern sunk in the floor, fed by a lead pipe from
  ↪ somewhere
  above. It is a block of grey ice to its brim, and something flat
  ↪ and
  pale is held in the ice a hand's depth down.
```

```
create the tower bell
  aka bell
  a thing, scenery
  in the Bell Stair
  states: unrung, tolling, silent

  You cannot see the bell from here, only the dark it hangs in.
```

```
create the spark page
  aka first page, leaf
  a thing, readable with text "One line, in ink that is still wet.
  ↪ It is a single word, and it will not stay in your head."
  on the reading desk
  states: legible, learned

  A leaf that has worked loose from the grimoire. The ink on it is
  complete.
```

The cistern, the bell and the page are new. Each list tells a small story in order: the ice thaws and then runs, the bell has never rung, then tolls, then falls silent, and the page can be read before it has been learned.

13.2 Naming a state

A state names what the thing is. It does not name what the thing has stopped being. If you give a wax seal the states `sealed, unsealed`, the compiler refuses the second word with `analysis.negated-state`, because `unsealed` describes the absence of the first state and nothing else. Ask what the seal is once it is no longer sealed. It is `broken`, or `melted`, or `lifted`, and any of those tells the player and the next author more than the negative does.

13.3 Reading a state

A state is read with the word `is`. The phrase `the iron grille is standing` is a **condition**, which is to say something the story can test and find true or false. Conditions have a chapter of their own, which is chapter 16, but you met one place that takes a condition in chapter 12. A blocked way may be blocked only `while` something holds.

```
create the Bell Stair, a room
  in the Hall
  up to the Lantern Gallery
  up is blocked while the iron grille is standing: stair-sealed
```

In Part II the stair was sealed for good. Now it is sealed for as long as the grille is in its first state, and the line and the thing are tied together. To say that a thing is not in a state, write `is not: the iron grille is not standing`.

13.4 Changing a state

A state is changed with a **statement**, which is one line of instruction to the story.

```
change the iron grille to lifted
```

Statements cannot stand loose in a file. They live inside blocks that say when they run, and the block you will use most is the clause of chapter 14. There is one such block in the story already, the start block, and it will do for an experiment. The line needs the grille's `states` line, so try it once this chapter's story is in your folder, after the copy in "Build and try it" below. Add the line to the start block for a moment.

```
before the game starts
  change the player to Ansel
  change the iron grille to lifted
end before
```

Build the story and walk to the Bell Stair.

```
> up
Lantern Gallery
The roof of the tower, open on every side. The Great Lantern
stands in the middle of it on a stone drum, taller than you are,
its glass thirty years dirty. To the east a low parapet is all
there is between you and the court. The stair goes back down.
```

The stair is open, because the condition on the blocked line is no longer true. Take the extra line out again before you go on. Keep the trick in mind, though. The start block is a good place to put the world into the state you want to test, without playing your way there each time.

If you ask for a state the thing does not have, the compiler tells you.

```
probe.story:27:3 error [analysis.undeclared-state] `melted` is not
a declared state of seal.
```

13.5 States that go both ways

Nothing in Lanternhall goes back to what it was, which is rather the point of the story. For a thing that does, here is a sluice from another one.

```
create the sluice
  a thing, scenery
  in the Yard
  states, reversible: raised, lowered

  A sluice.
```

The word `reversible` goes after `states` and before the colon. With it, the sluice may be changed from `raised` to `lowered` and back as often as the story likes. Without it, the second change is an error, and the message shows you the line to write:

```
probe.story:37:3 error [analysis.irreversible-state] `whole-shell`
is the initial state of a forward-only set — `change` cannot go
back. Declare the set as `states, reversible: whole-shell,
cracked-shell` to permit back-transitions.
```

Leave `reversible` off unless the thing really does go both ways. The forward-only rule will then catch the day you write a change in the wrong direction.

13.6 The story's own states

Some facts belong to no one thing. Lanternhall as a whole is asleep, and then it is waking, and at the end it is awake. The story header takes a `states` line of the same shape.

```
story
  title: The Relighting
  id: relighting
  story-version: 0.3.0
  states: asleep, waking, awake
```

The story's state is changed with `change the story to waking`, and it is read with `the story is asleep` or `the story is not asleep`. Everything said above about order and direction holds for the story as well.

Use the story's states for the few large facts that many things will ask about. A fact about one thing belongs on that thing. Chord has no loose variables and no flags, and that is by design. Every fact has an owner, and you find the fact by looking at its owner's block.

13.7 The states you do not declare

Some things already had states before this chapter. The library door is locked, the oak press is closed, and the hand lamp is unlit. Those come with the trait words of Part II, and they are read in the same way.

Trait word	What you can read
`openable`	`is open`, `is closed`
`lockable`	`is locked`, `is unlocked`
`switchable`	`is on`, `is off`
`light-source`	`is lit`
`wearable`	`is worn`
`breakable`	`is broken`

So `the library door is locked` is a condition you may already write, and `change the library door to unlocked` is a statement you may already write. For light, say `is lit` and `is not lit`. The word `unlit` belongs to the `starts unlit` line and is not something a condition reads:

```
probe.story:26:42 error [analysis.unknown-value] `unlit` is not a
state of lamp or a known trait — did you mean `lit`?.
```

Because the trait owns those words, you may not declare a state of the same name on a thing that has the trait.

```
probe.story:14:11 error [analysis.shadow-state] `open` is a word
of `openable`, which box carries — a state of that name would
answer `is open` in place of the trait. Rename the state, or read
and change the trait's own (`is open`, `change … to open`).
```

One trait word from chapter 9 makes its positions out of your states. A thing that is `settable` is a dial, and the states you declare are the marks on it.

```
create the dial
  a thing, scenery, settable
  in the Yard
  states, reversible: low, high

  A dial with two marks, low and high.
```

```
> set dial to low
You set the dial to low.

> set dial to middling
The dial goes to low or high.
```

13.8 Build and try it

Copy `relighting.story` from the companion folder `chapter-13` over your own file. It is the story with this chapter's changes made. To make the changes by hand instead, work from this list and take the rest from that file: add the `states` line to the header and to the grille and the lens, add the cistern, the tower bell and the spark page, put the `while` condition on the Bell Stair's blocked line, and raise `story-version` to `0.3.0`, because this is the first chapter of Part III.

```
sharpee compose relighting.story --check
sharpee play .
```

Play it, and you will find that nothing has changed. That is correct. Every thing is in its first state and no line yet moves one. The page can be read, and reading it teaches nothing.

```
> read page
(first taking the spark page)

You take the spark page from the reading desk.

The spark page reads:
One line, in ink that is still wet. It is a single word, and it
will not stay in your head.
```

UNDER THE HOOD: WHERE A STATE LIVES · STORY IR `states`

Compile the story with `sharpee compose relighting.story -o relighting.json` and look at the entry for the grille. Its `states` field holds the list in the order you wrote it, and a field beside it records whether the set is reversible. The trait states are not there. `is locked` and `is lit` are read from the platform's own lockable and light-source traits each time they are asked for, which is why they can never disagree with what the standard actions have done.

FROM ANOTHER SYSTEM: PROPERTIES AND FLAGS

In Inform 10 you would write "The cistern can be frozen, thawing or running" and then test and set that property anywhere in the source. In TADS 3 you would add a property to the object and assign to it from any method. Chord's `states` line does the same job with two differences. The states are ordered, and the order is enforced unless you say `reversible`. And there is no global variable to fall back on: a fact belongs to a thing or to the story, and it is declared in that owner's block.

13.9 What you have so far

The story and five of its things can now be in one condition or another, and one exit already depends on which. What is missing is anything that changes a state during play. For that a thing has to notice what the player does to it, and that is the next chapter.

14 Reacting to the player: on and after

Every command a player types becomes an **action**: taking, opening, reading, pulling. Chapter 9 called these the standard actions, and so far they have done all the work. The player types `take kettle` and the taking action takes the kettle.

A **clause** is how a thing gets a say in that. It is a small block inside a `create` block that names an action and holds the lines to run when the player does that action to this thing. There are two kinds. An `on` clause runs before the action and can stop it or speak in its place. An `after` clause runs once the action has succeeded and adds to it.

14.1 A thing that says no

Hobb's kettle has been sitting in the Lodge since chapter 6, and any player could walk off with it. It belongs to Hobb, and Hobb is four feet away.

```
create the kettle
  a thing
  in the Lodge

  A black iron kettle, still warm.

  on the player taking while Hobb is here
    refuse kettle-is-hobbs
  end on
```

Read the first line of the clause as a sentence: on the player taking, while Hobb is here. The word after `the player` is a **gerund**, the "-ing" form of a verb, and it names the action. The clause belongs to the kettle because it is written in the kettle's block, so it concerns the player taking the kettle and nothing else.

The line inside is a statement. `refuse` stops the action and prints a phrase, and the phrase goes in the `define phrases` block with the others.

```
define phrases en-US
  kettle-is-hobbs:
    Hobb does not look round. "That's mine," he says.
```

The clause closes with `end on`.

```
> take kettle
Hobb does not look round. "That's mine," he says.
```

The kettle is still on the floor of the Lodge. The taking action never ran.

14.2 Where a clause lives

A clause is written on the thing it is about. This is the most important habit in Chord, and the language holds you to it. There is no place in a story file for a rule that says “when the player takes the kettle”. There is only the kettle, and what the kettle does about being taken.

The gain shows as a story grows. Everything that can happen to the kettle is in the kettle’s block, and to find out why the grille will not move you read the grille.

14.3 Speaking in the action’s place

An `on` clause does not have to refuse. If it prints a phrase and lets the action go on, the phrase replaces what the action would have said.

The spark page was `readable with text` in the last chapter, and reading it printed the standard sentence followed by that text. Take the text away and give the page a clause.

```
create the spark page
  aka first page, leaf
  a thing, readable
  on the reading desk
  states: legible, learned

  A leaf that has worked loose from the grimoire. The ink on it is
  complete.

  on the player reading
    change the spark page to learned
    phrase spark-learned
  end on
```

```
define phrases en-US
  spark-learned:
    You read the line once, and it reads you back. The word for
    ↪ waking a
    flame settles into the place it left thirty years ago, as if
    ↪ it had
    only stepped out.
```

The clause has two statements. The first is the `change` you met in the last chapter, now in a place where it runs during play. The second, `phrase`, prints a piece of text.

```
> read page
(first taking the spark page)

You take the spark page from the reading desk.

You read the line once, and it reads you back. The word for waking
a flame settles into the place it left thirty years ago, as if it
had only stepped out.
```

The page's state is now `learned`. Something in Lanternhall has changed because of what the player did, for the first time.

14.4 Adding to an action: after

An `after` clause leaves the action alone and speaks once it is done. The bell rope is `pullable`, and pulling it prints the standard sentence. The rope then adds its own.

```
create the bell rope
  aka rope
  a thing, scenery, pullable
  in the Bell Stair

  A rope as thick as your wrist, coming down out of the dark.

  after the player pulling
    phrase rope-nothing
  end after
```

```
> pull rope
You pull the bell rope.

Nothing answers. The bell has not been rung in thirty years and
sees no reason to start for you.
```

Both lines are printed, the action's first and the clause's second. An `after` clause closes with `end after`.

Because it runs when the action has already happened, an `after` clause cannot refuse. If you write `refuse` in one, the compiler explains:

```
probe.story:23:5 error [parse.react-refusal] Refusals (`refuse`,
`must`) cannot appear in an `after` clause — reactions run after
the action succeeded. Use an `on` clause to intercept.
```

The choice between the two is the choice between those two sentences. Use `on` when the thing should prevent the action or tell it differently. Use `after` when the action should happen as usual and the thing has something to add.

14.5 Several actions, one clause

The grille cannot be taken, pushed or pulled, and it says the same thing to all three. A clause head may list gerunds, separated by commas.

```
create the iron grille
  aka grille
  a thing, scenery
  in the Bell Stair
  states: standing, lifted

  Wrought iron, floor to vault, with no lock, no latch and no
  ↪ hinges.
  It was not built. It was put there.

  on the player taking, pushing, pulling
    refuse grille-fixed
  end on
```

```
> push grille
It does not so much as rattle.

> pull grille
It does not so much as rattle.
```

A list is a shorthand for one clause per gerund, each with the same lines. It is not a way to have two clauses for one action. A thing takes one `on` clause and one `after` clause for each gerund, and a second is an error:

```
probe.story:26:17 error [analysis.duplicate-clause] A `on taking`
clause is already declared on lamp at line 22 — a second one would
silently mask it. Merge the bodies (or split intercept/react
across `on`/`after`).
```

When a thing needs to answer one action in more than one way, it does so inside one clause. Chapter 17 shows how.

14.6 The other thing in the command

Some actions involve two things. The player gives the spectacles to Hobb, or puts the lens in the Lantern. A clause on one of them may name the other, and then it runs only for that pairing.

```
create the spectacles
  aka glasses
  a thing, plural, concealed
  in the Lodge

  Wire spectacles with one arm mended in thread. Hobb has been
  ↪ looking
  for them since the spring.

  after the player giving to Hobb
    phrase spectacles-given
  end after
```

```
> give spectacles to hobb
You give the spectacles to Hobb.

Hobb hooks them over his ears and looks at you properly for the
first time. "Ansel," he says, after a while. "You were always
late."
```

The words `to Hobb` are the **counterpart**. Giving the spectacles to anyone else would not run the clause. The lens has one of the same kind, for the day it reaches the roof:

```
  after the player inserting in the Great Lantern
    phrase lens-seated
  end after
```

Two gerunds are easy to confuse here. `putting` is putting a thing on a supporter, and `inserting` is putting a thing in a container. The Great Lantern is a container, so the lens is inserted. Appendix C lists every standard action with its gerund.

A `settable` thing has a tail of a different kind. Its clause may name the position being chosen, and then it runs for that position only.

```
  on the player setting to high
    phrase dial-high
  end on
```

```
> set dial to high
The needle swings over to high, and something below you starts to
hum.
```

14.7 Conditions and once

A clause head may end with `while` and a condition, as the kettle's does. The clause then runs only when the condition is true. If Hobb ever leaves the Lodge, the kettle can be taken.

A head may also end with a comma and the word `once`. The clause then runs one time in the whole story and never again.

```
create the Library, a room
  in the Hall

  The reading room of Lanternhall, with its desks in rows and its
  galleries of shelves. The stacks run back to the north, out of
  ↪ the
  light. The cloister is east.

  after the player entering, once
    phrase library-ink
  end after
```

```
> west
Library
The reading room of Lanternhall, with its desks in rows and its
galleries of shelves. The stacks run back to the north, out of the
light. The cloister is east.

On the reading desk you see a grimoire and a spark page.

There is a smell in here that was not here thirty years ago, or
was here and you had forgotten it. It is wet ink.
```

This clause is on a room, and on a room the gerund `entering` means the player's arrival there. On anything else it means the player getting into the thing, as with Hobb's chair.

14.8 Clauses about the person

Every clause so far has been about something done to the thing that owns it. A person's block may also hold a clause about what that person does. The head then has no actor, only the gerund.

```
create Ansel
  a person, proper
  playable
  starts in the Lodge
  carries the wax taper
  wears the travelling coat

  You were a student here once, for a term. You have the same
  ↪ hands and
  not much else.

  after eating
    phrase cake-eaten
  end after
```

This runs after Ansel eats anything at all. The only food in Lanternhall is Hobb's cake.

```
> eat cake
(first taking the seed cake)

You take the seed cake from the chair.

You eat some of the seed cake.

It tastes of caraway, and of being nineteen.
```

A head with no actor is allowed only in a person's block. The actor in the other heads does not have to be the player, either. `after Hobb taking` would be a clause about Hobb taking the thing, and it waits for chapter 26, where people act by themselves.

14.9 Build and try it

Copy `relighting.story` from the companion folder `chapter-14` over your own file. It is the story with this chapter's changes made, and it holds the phrases, which the chapter did not print in full. To make the changes by hand instead, work from this list and take the phrases from that file: add the clauses to the kettle, the spectacles, the Library, the spark page, the bell rope, the grille, the lens and Ansel, add their eight phrases, and remove `with text` and its string from the spark page.

```
sharpee compose relighting.story --check
sharpee play .
```

Find the spectacles first. They are `concealed`, so the command is `search`.

```
> search
Hidden here, you discover: spectacles.

> take spectacles
Taken.
```

UNDER THE HOOD: THE PHASES OF AN ACTION · `IF.ACTION.TAKING`

A standard action runs in phases. It first checks whether it may happen, then changes the world, then reports what it did. An `on` clause is consulted at the start, before the check, which is why it can refuse and why a phrase it prints stands in for the report. An `after` clause runs when the report is already written. `sharpee test` lists every clause it found by owner and gerund, such as "kettle · on taking", and that list is a quick way to see what your story has attached to which action. The command needs a test tree beside the story, which chapter 38 teaches you to write, and your own folder has none yet. The companion folder `chapter-14` has one. Go into that folder and run `sharpee test .` to see the list, under the heading that begins "Derived rule tests".

FROM ANOTHER SYSTEM: RULES ABOUT ACTIONS

Inform 10 gives you rulebooks that sit around every action: Instead, Before, Check, Carry out, After and Report rules, each of which may name the object it applies to. TADS 3 puts the handling on the object, in `dobjFor` blocks with `verify`, `check` and `action` methods. Chord is closer to the second. It has two hooks where Inform has six, and it will not let you write either of them anywhere but on the thing concerned.

14.10 What you have so far

Lanternhall answers back. A kettle refuses, a page teaches, a rope and a room add a sentence, and the porter knows your name. Each of those

clauses holds one or two statements, and you have met three statements in all: `refuse`, `phrase` and `change`. The next chapter is about the rest.

15 What a clause can say and do

A clause is a list of statements, run from the top. The last chapter used three of them. This chapter goes through the rest of the everyday ones: the two ways to refuse, the statements that print, change and move, and the suffix that makes any of them conditional.

Chord has no `if`. There is no line that says "if this, do these three things, otherwise do those". It is the first thing an author from another language looks for, and the reason for its absence is the subject of half this chapter. What Chord has in its place is a set of smaller forms, each of which says one thing plainly.

15.1 Requirements: must

Ansel remembers the shape of the old workings and none of their words. The hand lamp should not light until the first word has been read. That is a **requirement**, a thing that has to be true before the action may go on, and it is written with `must`.

```
create the hand lamp
  a thing, light-source, starts unlit
  in the Lodge

  A tin hand lamp with a wire handle. Its wick is dry and its
  ↪ glass is
  clean, which is Hobb all over.

  on the player lighting
    the spark page must be learned: no-such-word
  end on
```

The line states what must hold, then a colon, then the phrase to print when it does not. If the spark page is `learned`, the line does nothing and the clause goes on. Here there is nothing more in the clause, so the

standard lighting action runs as usual. If the page is not learned, the action is refused with the phrase.

```
> light lamp
You reach for the word and find only the place where it was.
```

A `must` line can ask the same things a condition can, in the form that follows `must`:

Form	Example
`must be` a state	`the spark page must be learned`
`must be in` a place	`the lens must be in the Great Lantern`
`must be a` kind	`the crate must be a container`
`must have`, `must hold`, `must wear`	`the player must hold the hand lamp`
`must see`, `must reach`	`the player must reach the lens`

The Great Lantern has two requirements, and they are checked in order.

```
on the player lighting
  the spark page must be learned: no-such-word
  the lens must be in the Great Lantern: lantern-empty
end on
```

A player who has not read the page is told about the word. A player who has read it and brought no lens is told about the lens. The first requirement that fails is the one that speaks.

15.2 Prohibitions: refuse when

A requirement says what must be so. Its opposite is a **prohibition**, which names something that stops the action when it is present. Hobb's cake may be eaten, but not while Hobb is watching.

```
create the seed cake
  aka cake
  a thing, edible
  on the chair
  servings 2

  Half a seed cake on a plate, cut with some care into two pieces.

  on the player eating
    refuse when Hobb is here: cake-watched
  end on
```

```
> eat cake
Hobb cut that cake into two pieces with some care, and he is
sitting four feet away. You find you cannot.
```

Carry the cake out to the Gate Court and it can be eaten, because Hobb is not there.

Chord keeps these two forms apart, and it will not let you turn one into the other by adding `not`. Both of the following are errors.

```
probe.story:23:19 error [parse.must-negative] State requirements
positively — `must not …` is not a form. Use `refuse when
<condition>: <key>` for prohibitions.
```

```
probe.story:23:5 error [analysis.negated-requirement] State
requirements positively — `refuse when not …` is not a form. Write
`<subject> must <predicate>: <phrase-key>` for the requirement;
`refuse when` is for hazards that are present.
```

The rule sounds fussy and reads well. A `must` line always tells you what the player needs. A `refuse when` line always tells you what is in the way. You never have to untangle a negative to find out which.

The third form is the plain `refuse` of the last chapter, which stops the action without asking anything. All three belong at the top of an `on`

clause, above the statements that do things, and none of them may appear in an `after` clause.

15.3 Printing: phrase

`phrase` prints the text that a key stands for. The key is declared in the `define phrases` block, or it may be declared on the spot, by writing the text beneath the statement and indenting it one level further.

```
after the player touching
  phrase cat-goes
    The cat is elsewhere, at once.
end after
```

A phrase written in place belongs to that clause and cannot be used from another one. It suits a line that is said once. Text that several clauses share, such as `no-such-word`, belongs in `define phrases`, and so does anything you will want to find again when you revise the prose.

15.4 Changing and moving

`change` sets a state, and you have used it. The statements that move things are these.

Statement	What it does
`move the page to the Refectory`	puts a thing in a room, in a container, on a supporter or in a person's hands
`move the page here`	puts it where the player is
`move the cat offstage`	takes it out of the world, to be brought back by a later `move`
`remove the bun`	takes it out of the story for good
`make Hobb wear the spectacles`	puts a wearable thing on a person

Statement	What it does
`make the player take off the robe`	takes it off, and leaves it in their hands

A thing may also begin offstage. If a `create` block has no placement line, the thing exists and is nowhere, and waits for a `move`. The refectory candles use this. Somewhere under the middle pricket there is a page that daylight does not show.

```
create the unbind page
  aka fourth page
  a thing, readable with text "It reads like a knot seen from the
  ↪ wrong end."

  A leaf of the grimoire with a ring of old wax on it, where a
  ↪ pricket
  stood for thirty years.
```

```
create the candles
  aka prickets
  a thing, scenery, plural, light-source, starts unlit
  in the Refectory

  Stubs of tallow in iron prickets down the middle of each table.
  ↪ Nobody
  has put a flame to them since the last supper of the last term.

  on the player lighting
    the spark page must be learned: no-such-word
  end on

  after the player lighting
    move the unbind page here
    phrase candles-catch
  end after
```

The candles have an `on` clause and an `after` clause for the same action, which is allowed and often wanted. The first decides whether lighting may happen. The second says what follows when it has.

```
> light candles
You light the candles.

The flame runs down the length of the table, and every cold wick
takes it. In the candlelight there is a leaf of paper under the
middle pricket that the daylight did not show.

> look
Refectory
Three long tables, each with its row of candles, and a hearth you
could stable a horse in. The benches are stacked against the wall,
where someone left them for the holidays. The cloister is back to
the west.

You can see an unbind page here.
```

Hobb's spectacles get one more line. Giving them to him already printed a phrase, and now he puts them on.

```
after the player giving to Hobb
  make Hobb wear the spectacles
  phrase spectacles-given
end after
```

For this the spectacles need the trait word `wearable` on their kind line. The statement prints nothing by itself. The phrase does the telling.

15.5 The when suffix

Any statement that does something may end with `when` and a condition. The statement then runs only if the condition is true.

The spark page should behave differently the second time it is read. The first reading teaches the word and wakes the Hall. Later readings should only say that Ansel knows it.

```
  on the player reading
    phrase spark-known when the spark page is learned
    phrase spark-learned when the spark page is legible
    change the story to waking when the spark page is legible
    change the spark page to learned
  end on
```

```
> read page
(first taking the spark page)

You take the spark page from the reading desk.

You read the line once, and it reads you back. The word for waking
a flame settles into the place it left thirty years ago, as if it
had only stepped out.

> read page
You know it. You are not likely to lose it twice.
```

This is what Chord offers where another language would offer `if`, and you should see its two costs.

Each line carries its own condition, so a clause with two outcomes repeats the condition on every line.

The lines also run in order, and an earlier one can change what a later one tests. The `change the spark page to learned` line is last for that reason. Move it to the top and both phrases would find the page `learned`, and the first reading would print the wrong one.

For a clause with one or two conditional lines, the suffix is the right tool. For a clause that really has branches, chapter 17 has a better one, and it will rewrite this clause.

`refuse` takes no `when` suffix. Its conditional form is `refuse when`, which you have already.

15.6 It, and its absence

In every clause above, the thing is named in full: `the spark page`, `the hand lamp`. You might expect to write `it`. In a `create` block you may not.

```
probe.story:23:17 error [analysis.it-removed] `it` is no longer a
form outside `define trait` — name the owner: `the lamp`.
```

A clause in a thing's block often mentions two or three things, and `it` would leave the reader to work out which one is meant. The word comes back in chapter 24, in the one place where the thing has no name to use.

15.7 Build and try it

Copy `relighting.story` from the companion folder `chapter-15` over your own file. It is the story with this chapter's changes made, and it holds the phrases, which the chapter did not print. To make the changes by hand instead, work from this list and take the rest from that file: add the clauses to the hand lamp, the seed cake, the Great Lantern and the candles, give the Lantern and the candles `light-source, starts unlit` and the spectacles `wearable`, rewrite the spark page's clause, add the unbind page, and add the five new phrases.

```
sharpee compose relighting.story --check
sharpee play .
```

The first steps of the game now have to be done in order. The lamp will not light until the page is read, the page is behind a locked door, and the key is in the oak press.

```
> light lamp
You light the hand lamp.

> down
Undercroft
Low vaults on squat pillars, running off under the whole of the
hall. The air is cold and smells of stone. The steps go back up.

You can see a crate here.
```

UNDER THE HOOD: WHAT THE TEST RUNNER DERIVES · `SHARPEE TEST`

Run `sharpee test .` inside the companion folder `chapter-15`, which has a test tree where your own folder has none, and read the list under the heading that begins "Derived rule tests". The report has other parts above and below that list, and chapter 38 explains them. The runner has turned each refusal into a test of its own: "hand lamp · on lighting · refused no-such-word", "seed cake · on eating · refused cake-watched". It arranges the world so that the refusal should fire, types the command, and checks that the phrase came out. A `must` line and a `refuse when` line are both stored as a condition and a phrase key, which is all the runner needs to do that.

FROM ANOTHER SYSTEM: NO IF

Both Inform 10 and TADS 3 let you put a conditional block wherever statements go. Chord does not. A refusal is a `must` or a `refuse when` at the top of the clause, a single conditional effect takes a `when` suffix, and a real branch is a `select` block from chapter 17. If you find yourself wanting three or four `when` suffixes with the same condition, that is the language telling you the thing has a state you have not named yet.

15.8 What you have so far

A clause can now require, forbid, print, change and move. The lamp wants a word, the candles hide a page, and the story itself has moved from `asleep` to `waking`. Every one of those lines leans on a condition, and so far the conditions have been as simple as they come. They are the next chapter.

16 Conditions

A condition is anything the story can test and find true or false. You have been writing them since chapter 12: after `while` on a blocked way, after `while` on a clause head, after `when` on a statement, and inside `must` and `refuse when`. All of those places take the same thing. This chapter is the whole of what that thing can be.

16.1 What a condition can ask

The simplest condition has a subject and one test.

Test	Example	True when
`is` a state	`the lens is sound`	the thing is in that state
`is a` kind	`the crate is a container`	the thing is of that kind
`is in`	`the lens is in the Great Lantern`	the thing is directly in or on the other
`is here`	`Hobb is here`	the thing is in the room where the player is
`has`	`the player has the lamp`	the person is carrying or wearing it
`holds`	`the player holds the lamp`	the same as `has`
`wears`	`the player wears the coat`	the person has it on
`can see`	`the player can see the lens`	the thing is in view
`can reach`	`the player can reach the lens`	the thing can be touched

Three of these need a note.

`is in` is used for supporters too. A cup on the chair is `in the chair` as far as a condition is concerned, and there is no `is on`.

`is here` is generous. A thing the player is carrying is here, and so is a thing shut inside a container in the room.

`can see` and `can reach` differ in the way you would hope. A coin in a closed glass jar can be seen and cannot be reached. Open the jar, and it can be both.

Any `is` test is made negative with `not` after the `is`: `the lens is not sound`, `Hobb is not here`.

There is one test of a different kind.

```
phrase rooks-go-up when one chance in 3
```

`one chance in 3` is true about a third of the times it is asked. Use it for colour and never for anything the player needs, because a player cannot be expected to repeat a command until the dice agree.

16.2 Joining conditions

Conditions are joined with `and` and `or`, and turned round with `not`.

```
the lens is in the Great Lantern and the lens is sound
```

`not` is applied first, then `and`, then `or`. When that is not the grouping you mean, use parentheses.

```
(the jar is open or the player wears the hat) and not the player
↪ has the coin
```

A whole condition goes on one line. If a line is getting long, that is a sign the condition wants a name.

16.3 Naming a condition

The Great Lantern is ready to be lit when the lens is seated in it and the lens is whole. Several lines of the story will want to ask that, so it is given a name once, at the top level of the file.

```
define condition lantern-ready: the lens is in the Great Lantern
↪ and the lens is sound
```

From then on the bare word `lantern-ready` is a condition, and may be used anywhere a condition goes: `when lantern-ready`, `when not lantern-ready`, `while lantern-ready and the story is waking`.

A named condition is worth having for three reasons. It is shorter. It says what the test means, where the long form says only how it is made. And when the rule changes, because the Lantern turns out to need a shutter as well, you change it in one place.

A named condition is worked out afresh each time it is asked. It stores nothing. It is a name for a question, and the answer is always current.

16.4 Conditions with a gap: it

The pages of the grimoire did not all come back at once. The spark page is legible this morning. The other pages are still faded, and the ink returns to them when the first working is read. So two more pages get a new first state.

```
create the hush page
  aka third page
  a thing, readable
  in the Stack 3
  states: faded, legible, learned

  A leaf of the grimoire, shelved by somebody between two blank
  ↪ spines
  as though it were a book.
```

The unbind page under the candles gets the same `states` line. Now the story needs to talk about every faded page, whichever pages those are and wherever they have got to. A condition can leave its subject open, with the word `it`.

```
define condition faded-page: it is faded
```

This is a condition with a gap in it. By itself it cannot be true or false, because it does not say what `it` is. It is a description, and things either fit it or do not. The hush page fits it. The kettle does not.

An open condition is used in three ways.

16.5 Any and no

`any` followed by the name is true when at least one thing fits. `no` followed by the name is true when nothing does.

```
create the grimoire
  aka book, margin, note
  a thing, readable with text "In the margin, in a small brown
  ↪ hand: gnusto. Came over the water as a working, they say.
  ↪ Meaning lost."
  on the reading desk

  A book as long as your forearm, chained to nothing any more. Its
  ↪ pages
  are blank, or were. Here and there a line of ink has come back,
  ↪ as
  frost comes onto a window. Someone has written in one margin.

  after the player examining
    phrase grimoire-waiting when any faded-page
    phrase grimoire-filling when no faded-page
  end after
```

```
> examine grimoire
A book as long as your forearm, chained to nothing any more. Its
pages are blank, or were. Here and there a line of ink has come
back, as frost comes onto a window. Someone has written in one
margin.

The loose leaves that belong in it are somewhere about the Hall,
and you have the feeling they are still as blank as these.
```

The grimoire does not know how many pages there are or what they are called. It asks whether anything in the story is faded.

16.6 Each

The third use does something to every thing that fits. An `each` block is written inside a clause. It names an open condition, and its lines run once for each thing the condition holds for. Inside the block, the words `the match` stand for that thing.

Reading the spark page is what brings the ink back.

```
on the player reading
  phrase spark-known when the spark page is learned
  phrase spark-learned when the spark page is legible
  change the story to waking when the spark page is legible
  change the spark page to learned
  each faded-page
    change the match to legible
  end each
end on
```

The block closes with `end each`. When the player reads the spark page, the hush page and the unbind page both become `legible`, in the same turn, wherever they are. If a later chapter adds a fifth page with a `faded` state, this clause will wake it too, and nobody will have to remember to come back and edit it.

That is the difference between `each` and writing two `change` lines. The two lines name what is in the story today. The block says what you mean.

A faded page should not be readable, and that is a prohibition on the page.

```
on the player reading
  refuse when the hush page is faded: page-blank
  phrase hush-known when the hush page is learned
  phrase hush-learned when the hush page is legible
  change the hush page to learned
end on
```

A player who goes straight to the stacks finds the page and cannot read it.

```
> read hush page
The leaf is blank on both sides. You hold it up to the light, and
it is still blank, but the paper has the look of a held breath.
```

After the spark page has been read, the same command works.

```
> read hush page
(first taking the hush page)

Taken.

The third working is hardly a word at all. It is the breath you
take before one.
```

The grimoire has noticed as well.

```
> examine grimoire
A book as long as your forearm, chained to nothing any more. Its
pages are blank, or were. Here and there a line of ink has come
back, as frost comes onto a window. Someone has written in one
margin.

Something has changed in the last hour. Wherever its loose leaves
have got to, the ink is on them now.
```

`the match` may be used in the block wherever a thing's name could go, and only there. An `each` block may sit inside another, and `the match` then means the thing of the inner one.

16.7 What a condition cannot ask

A condition cannot ask who the player is. When a story has more than one person the player may be, as chapter 31 describes, the test is made on a state of the person concerned.

A condition cannot count. There is no "at least two faded pages". Counting is done with a counter, which chapter 18 introduces, and a counter can be compared with a number.

And a condition does not remember. `the lens is sound` asks about now. If the story needs to know that something was once true, that is a state, and some clause has to set it.

16.8 Build and try it

Copy `relighting.story` from the companion folder `chapter-16` over your own file. It is the story with this chapter's changes made, and it holds the phrases and the unbind page's clause, which the chapter did not print. To make the changes by hand instead, work from this list and take the rest from that file: add the two `define condition` lines near the top of the file, after the header, add the hush page, give the unbind page its states and its clause in place of `with text`, and add the `each` block to the spark page and the `after` clause to the grimoire. There are seven new phrases.

```
sharpee compose relighting.story --check
sharpee play .
```

Walk to the third stack before you read anything, try the hush page, and then come back for the spark page. The order matters now, and the story knows it.

> **UNDER THE HOOD: A CONDITION IS A TREE · STORY IR `CONDITIONS`**
>
> Compile to IR with chapter 4's command, `sharpee compose relighting.story -o relighting.ir.json`, and find `lantern-ready` under `conditions`. It is stored as an `and` with two branches, each a test with a subject and an object. Every `while`, `when` and `must` in the file holds a tree of the same shape, or the name of one. `faded-page` is marked `open`, which is the compiler's word for a condition with `it` in it, and the entries for `any`, `no` and `each` refer to it by name.

FROM ANOTHER SYSTEM: DESCRIPTIONS OF OBJECTS

Inform 10 has a close relative of the open condition. You write "Definition: a page is faded if …" and then say "if any page is faded" or "repeat with P running through faded pages". TADS 3 gives you loops over objects and leaves the filtering to ordinary code. Chord's `define condition`, `any`, `no` and `each` cover the same ground in four small forms. The limit is that an `each` block names a condition you have defined. You cannot write the test inline.

16.9 What you have so far

Conditions can be joined, named and left open, and one clause can now reach every thing of a sort. The reading clauses are getting crowded, though. Each has three or four lines ending in `when`, all about the same state. The next chapter gives them a proper shape.

17 Choosing between outcomes

The reading clauses of the last two chapters work, and they are hard to read. Four lines each end in `when`, the order of the lines matters, and the reader has to hold a state in their head to see which lines will run. The clause is really saying something simple: if the page is legible, do this, and if it is learned, do that.

Chord has three blocks for choosing. One chooses by a state. One chooses by turn, or by chance. One chooses by how many times the clause has run.

17.1 Choosing by state: select on

A `select on` block names a thing's state and gives one **arm** for each value you care about. Here is the spark page's clause again.

```
on the player reading
  select on the spark page's state
    when legible
      change the spark page to learned
      change the story to waking
      phrase spark-learned
      each faded-page
        change the match to legible
      end each
    when learned
      phrase spark-known
  end select
end on
```

The first line is `select on`, the thing with `'s state` after it. Each arm opens with `when` and a state name, and its statements are indented beneath it. The block closes with `end select`.

Exactly one arm runs: the one for the state the page is in when the block is reached. That removes both of the costs chapter 15 warned about. No condition is repeated, and the order of the statements inside an arm no longer matters to the choice, because the choice has been made before any of them runs. The `change the spark page to learned` line can now stand first, where a reader expects it.

You do not have to give an arm for every state. A state with no arm does nothing.

17.2 Refusals stay outside

The hush page has three states, and its `faded` state is a refusal. It is tempting to give it an arm.

```
    select on the hush page's state
      when faded
        refuse page-blank
```

The compiler will not have it.

```
relighting.story:390:9 error [analysis.refusal-misplaced] Refusal
inside a `select` arm — it never fires, because refusals are
decided before any branch runs. Move `refuse page-blank` to the
top of the clause, above the block.
```

Whether an action goes ahead is settled first, from the `must`, `refuse when` and `refuse` lines at the top of the clause. Only then do the statements run. So the refusal keeps its place at the top, and the block handles the two states that remain.

```
on the player reading
  refuse when the hush page is faded: page-blank
  select on the hush page's state
    when legible
      change the hush page to learned
      phrase hush-learned
    when learned
      phrase hush-known
  end select
end on
```

This is the shape of most clauses that do real work: what stops the action, then what the action does, by case.

17.3 One clause, several answers

Chapter 14 said that a thing has one clause for each gerund, and promised a way to answer one action differently at different times. This is it. The cistern in the Undercroft looks different in each of its three states, and it has one `after` clause for being examined.

```
create the cistern
  aka ice
  a thing, scenery
  in the Undercroft
  states: frozen, thawing, running

  A stone cistern sunk in the floor, fed by a lead pipe from
  ↪ somewhere
  above.

  after the player examining
    select on the cistern's state
      when frozen
        phrase cistern-frozen
      when thawing
        phrase cistern-thawing
      when running
        phrase cistern-running
    end select
  end after
```

The description now holds only what is always true of the cistern, and the clause adds the sentence that depends on its state.

```
> examine cistern
A stone cistern sunk in the floor, fed by a lead pipe from
somewhere above.

It is a block of grey ice to its brim. Something flat and pale is
held in the ice, a hand's depth down.
```

Nothing thaws the cistern yet, so the other two arms wait for chapter 19.

17.4 Choosing by turn: select with a strategy

A response that never varies wears thin. The bell rope says the same sentence every time it is pulled, and a player will pull it more than once. A `select` block may be given a **strategy** in place of `on`, and then its arms are alternatives, separated by lines that say `or`.

```
  after the player pulling
    select cycling
      phrase rope-nothing
    or
      phrase rope-dust
    or
      phrase rope-patience
    end select
  end after
```

`cycling` takes the arms in order and starts again after the last.

```
> pull rope
You pull the bell rope.

Nothing answers. The bell has not been rung in thirty years and
sees no reason to start for you.

> pull rope
You pull the bell rope.

A little dust comes down the well, and that is all.

> pull rope
You pull the bell rope.

The rope takes your weight and gives nothing back. Somewhere
above, the bell goes on not ringing.

> pull rope
You pull the bell rope.

Nothing answers. The bell has not been rung in thirty years and
sees no reason to start for you.
```

There are five strategies.

Strategy	Which arm runs
`cycling`	each in order, then round again
`stopping`	

Strategy	Which arm runs
	each in order, then the last one from then on
`randomly`	any one, chosen afresh each time
`sticky`	one chosen at random the first time, and that one ever after
`first-time`	the first arm once, and the second arm from then on

`stopping` suits a thing that runs out of things to say. `sticky` suits a detail that should differ from one playing to the next and stay fixed within one. `first-time` takes two arms and is the plainest way to say a thing once and something shorter afterwards.

An arm may hold any statements, not only a `phrase`. One arm could move a thing and another could change a state. Most of the time, though, a strategy is being used to vary text, and chapter 20 has a shorter way to write that, with the variants inside the phrase.

17.5 Choosing by count: first time, second time

The third block counts. Inside a clause, `first time` opens a block that runs the first time the clause runs, `second time` the second, and so on up to `tenth time`.

```
create the iron grille
  aka grille
  a thing, scenery
  in the Bell Stair
  states: standing, lifted

  Wrought iron, floor to vault, with no lock, no latch and no
  ↪ hinges.
  It was not built. It was put there.

  on the player taking, pushing, pulling
    refuse grille-fixed
  end on

  after the player examining
    first time
      phrase grille-memory
    second time
      phrase grille-cold
    third time
      phrase grille-enough
  end after
```

```
> examine grille
Wrought iron, floor to vault, with no lock, no latch and no
hinges. It was not built. It was put there.

There was no grille here in your day. You came down this stair at
a run, most mornings.

> examine grille
Wrought iron, floor to vault, with no lock, no latch and no
hinges. It was not built. It was put there.

The iron is colder than the morning accounts for.

> examine grille
Wrought iron, floor to vault, with no lock, no latch and no
hinges. It was not built. It was put there.

It has not changed its mind.

> examine grille
Wrought iron, floor to vault, with no lock, no latch and no
hinges. It was not built. It was put there.
```

After the third time the blocks are used up, and the grille has only its description. These blocks need no `end` line. Each one closes where the next begins, or where the clause does.

You met `first time` in chapter 5, as the text a room shows on the first visit. It is the same idea. On a room it stands in the block by itself. In a clause it counts that clause.

17.6 Which one to use

The three blocks answer three different questions, and it is worth asking which question you have before you write one.

If the answer depends on what is true of the world, use `select on`. The cistern does not thaw because it has been looked at three times.

If the answer should simply vary, use a strategy. Nothing about the bell rope has changed between one pull and the next.

If the answer depends on the player having done this before, use the count. The grille's third response is a small joke about persistence, and it is funny only as the third.

When the count starts to matter to the story, as when the fifth pull of a rope should bring the bell down, you have gone past what these blocks are for. That is a number the story should keep, and it is the next chapter.

17.7 Build and try it

Copy `relighting.story` from the companion folder `chapter-17` over your own file. It is the story with this chapter's changes made, and it holds the phrases and the unbind page's clause, which the chapter did not print. To make the changes by hand instead, work from this list and take the rest from that file: rewrite the reading clauses of the spark, hush and unbind pages with `select on`, move the frozen sentence out of the cistern's description and give the cistern its clause, and put the `select cycling` block in the bell rope's clause and the three counted blocks on the grille. There are eight new phrases.

```
sharpee compose relighting.story --check
sharpee play .
```

UNDER THE HOOD: A BRANCH IS A TEST · `sharpee test`

The derived tests of chapter 15 follow the branches. Run `sharpee test .` inside the companion folder `chapter-17` and you will see one line for each arm: "spark page · on reading · when legible", "spark page · on reading · when learned", "iron grille · after examining · the third time". For a `select on` the runner puts the thing in the arm's state before it types the command, which is something it can do only because the states are declared. An arm you never reach in play can still be exercised here. Not every arm is. A line that begins with ◌ is an arm the runner could not arrange, with its reason, and the cistern's three arms are listed that way in this chapter's story, because the runner could not bring the cistern into the player's sight.

FROM ANOTHER SYSTEM: VARYING TEXT

Inform 10 writes variation inside the text, with substitutions such as "[one of]…[or]…[cycling]" and "[first time]…[only]". TADS 3 has event lists, among them `ShuffledEventList` and `StopEventList`, attached to an object. Chord's strategies are the same family, and its names will be familiar from both. The difference is that a Chord `select` chooses between runs of statements, so the alternatives can do things as well as say them.

17.8 What you have so far

A clause can now branch on a state, rotate through alternatives and count its own visits. The reading clauses say what they mean. Part III has two chapters left, and they add the two things a puzzle game is usually built on: numbers and time.

18 Counting, scoring and ending

A state says which of a few named conditions a thing is in. Some facts are not like that. How many times the rope has been pulled, how many workings Ansel has back, how well the player is doing: these are numbers. This chapter covers the numbers a story keeps, the score built on them, and the statements that bring a story to its end.

18.1 A counter on a thing

A **counter** is a whole number with a name. A thing gets one with a line in its block.

```
create the bell rope
  aka rope
  a thing, scenery, pullable
  in the Bell Stair
  counter pulls

  A rope as thick as your wrist, coming down out of the dark.

  after the player pulling
    raise the bell rope's pulls by 1
    phrase rope-sore when the bell rope's pulls is more than 3
    select cycling
      phrase rope-nothing
    or
      phrase rope-dust
    or
      phrase rope-patience
    end select
  end after
```

The line `counter pulls` gives the rope a counter that starts at nought. From inside a clause it is called `the bell rope's pulls`, the owner and then the name. `raise` adds to it and `lower` takes away.

A counter is compared with a number in words or in symbols. Both lines below mean the same.

```
    phrase rope-sore when the bell rope's pulls is more than 3
    phrase rope-sore when the bell rope's pulls > 3
```

The word forms are `is at least`, `is more than`, `is at most` and `is less than`. The symbols are `>=`, `>`, `<=` and `<`.

```
> pull rope
You pull the bell rope.

Your palms have begun to object.

Nothing answers. The bell has not been rung in thirty years and
sees no reason to start for you.
```

That is the fourth pull. The counted blocks of the last chapter could have done this much. A counter earns its place when the number is read somewhere else, or goes down as well as up, or has limits.

The line may give a starting value and limits.

```
  counter sips starts 3 between 0 and 3
```

A counter with limits stops at them. Lowering `sips` when it is already at nought leaves it at nought. And `set` puts a counter at a value outright: `set the vial's sips to 3`.

18.2 A counter for the story

A number that belongs to no one thing is declared at the top level.

```
define counter workings
```

This one counts the workings Ansel has learned. It has no owner, so it is called by its bare name, and each page raises it in the arm where the page is first read.

```
      when legible
        change the hush page to learned
        raise workings by 1
        phrase hush-learned
```

The grimoire reads it.

```
  after the player examining
    phrase grimoire-waiting when any faded-page
    phrase grimoire-filling when no faded-page
    phrase grimoire-half when workings is at least 2
  end after
```

Chapter 16 said a condition cannot count. This is the way round it. The story counts as it goes, and the condition reads the count.

18.3 Scoring

A score is a number the player is shown. Chord builds it from **scores**, which are named awards with a value, each declared on the thing that earns it. The story header first says that the story keeps score at all.

```
story
  title: The Relighting
  id: relighting
  story-version: 0.3.0
  states: asleep, waking, awake
  use scoring
    rank "Latecomer" at 0
    rank "Second-Year" at 10
    rank "Lamplighter" at 25
```

`use scoring` turns the score on. It is the first `use` line in this book. A `use` line switches on a part of the language that not every story wants, and chapter 35 goes through the others.

An award is declared with a `score` line and paid with `award`.

```
create the spark page
  aka first page, leaf
  a thing, readable
  on the reading desk
  states: legible, learned
  score first-working worth 5
```

```
      when legible
        change the spark page to learned
        change the story to waking
        raise workings by 1
        award first-working
        phrase spark-learned
```

An award is paid once. If the `award` line runs again, nothing is added. The candles get `score supper-lit worth 5` and the Great Lantern gets `score relit worth 15`, each with an `award` line in the clause that earns it.

The player asks with `score`.

```
> score
You have scored 5 out of 25, earning you the rank of a Latecomer.
```

You did not write the 25. The maximum is the sum of every `score` line in the story, and it changes when you add one.

18.4 Ranks

The lines under `use scoring` are the **ranks**: a name in quotes, and the number of points at which the player reaches it. The rank is printed after the word “a”, so choose names that read well there.

When the player crosses into a new rank, the story says so.

```
> light candles
You light the candles.

The flame runs down the length of the table, and every cold wick
takes it. In the candlelight there is a leaf of paper under the
middle pricket that the daylight did not show.

You have risen to the rank of a Second-Year.
```

A rank’s number is a count of points and not a share of the maximum, which has a consequence. The compiler will not accept a rank that no player could reach.

```
relighting.story:19:5 error [analysis.rank-above-max] Rank
"Journeyman" sits at 30, above the 25 points this story declares —
no player could reach it.
```

The Relighting will end with four ranks and fifty points. For now it has twenty-five points, so it has three ranks, and the ladder will be lengthened as later chapters add awards.

18.5 Winning and losing

Two statements end the story.

```
win lantern-lit
lose lens-shatters
```

Each prints the phrase it names and then stops the game. `win` ends it in victory and `lose` in defeat.

Both endings of The Relighting belong to the Great Lantern. Lighting it over a sound lens is the victory. Lighting it over the cracked lens is the loud defeat. The named condition from chapter 16 keeps the clause readable.

```
create the Great Lantern
  aka lantern
  a container, scenery, light-source, starts unlit
  in the Lantern Gallery
  holds 1

  A drum of brass and glass the height of a door, with a seat
  ↪ inside it
  for a lens and a wick as thick as your arm. Nothing has burned
  ↪ in it
  for thirty years.

  score relit worth 15

  on the player lighting
    the spark page must be learned: no-such-word
    the lens must be in the Great Lantern: lantern-empty
  end on

  after the player lighting
    lose lens-shatters when not lantern-ready
    change the story to awake when lantern-ready
    award relit when lantern-ready
    win lantern-lit when lantern-ready
  end after
```

The `score` line stands after the description here, and among the short facts on the spark page. Either place will do.

Every line of the `after` clause carries a `when`, and that is not untidy writing. A `win` or a `lose` does not stop the clause it is in. The lines after it still run. If the `award` line had no condition, a player who had just

shattered the lens would be paid fifteen points for it. So in a clause that can end the story two ways, each line says which ending it belongs to.

18.6 Trying the endings

There is a difficulty. The Lantern is on the roof, the grille is across the stair, and nothing in the story can lift the grille until chapter 24. No player can reach either ending yet.

The start block trick from chapter 13 is made for this. Take a copy of the file as this chapter leaves it, which is the one “Build and try it” below copies in, since the endings are written there. In the copy, set the world up as it will be late in the game. Keep the copy, still named `relighting.story`, in a new folder beside your project called `relighting-endings`. It needs a folder of its own, since a project holds one story. Check and play it there.

```
before the game starts
  change the player to Ansel
  change the iron grille to lifted
  change the spark page to learned
  move the lens to the Lantern Gallery
end before
```

Then walk up to the Lantern Gallery, put the lens in the Lantern and light it.

```
> put lens in lantern
(first taking the lens)

Taken.

You put the lens in the Great Lantern.

The lens drops into its seat with a sound like a coin on a
counter.

> light lantern
You light the Great Lantern.

For one breath the cracked lens throws two beams, crossed, over
the whole valley. Then it lets go along the crack, and the Lantern
goes dark for the last time.

[The story has ended. RESTART, RESTORE or UNDO]
```

Add `change the lens to sound` as the last line of the start block and the same commands end the other way.

```
> light lantern
You light the Great Lantern.

The lens gathers the flame and sends it out over the valley in one
white bar. Down in the Hall, every blank page begins to fill.

You have risen to the rank of a Second-Year.

[The story has ended. RESTART, RESTORE or UNDO]
```

The companion file does not contain these start-block lines. They are a way of testing, and they come out again.

18.7 Killing the player

The third ending statement is for a death that is not a judgement on the whole game.

```
create the vial
  a thing, drinkable
  in the Yard

  A vial.

  after the player drinking
    kill the player
      It was not water. You have just long enough to be annoyed.
  end after
```

```
> drink vial
You drink the vial.

It was not water. You have just long enough to be annoyed.

[The story has ended. RESTART, RESTORE or UNDO]
```

`kill the player` takes a phrase key, or the text itself indented beneath it, and never both. It is what a `deadly` way from chapter 12 does for you. Use `lose` when the story has reached one of its endings and the player should know it. Use `kill the player` when they have simply walked into something, and everything chapter 12 said about fairness applies.

18.8 Build and try it

Copy `relighting.story` from the companion folder `chapter-18` over your own file. It is the story with this chapter's changes made, and it holds the phrases, which the chapter did not print. To make the changes by hand instead, work from this list and take the rest from that file: add `use scoring` and its ranks to the header, `define counter workings` near the top, the three `score` lines and their `award` lines, the `raise workings` line in each page's first arm, the counter on the bell rope, and the `after` clause on the Great Lantern. There are four new phrases.

```
sharpee compose relighting.story --check
sharpee play .
```

Ten of the twenty-five points can be earned in the story as it stands, five for the page and five for the candles.

```
> score
You have scored 10 out of 25, earning you the rank of a Second-
Year.
```

UNDER THE HOOD: THE SCORE IS A LEDGER · STORY IR `SCORES, RANKS`

In the compiled IR every award is one entry under `scores`, with its name and its worth, and the ranks are a list sorted by threshold. The platform keeps a ledger of which awards have been paid, and the number the player sees is the sum of that ledger. That is why an award cannot be paid twice, and why the maximum needs no line of yours: it is the sum of the list.

FROM ANOTHER SYSTEM: SCORING AND ENDINGS

Inform 10 keeps a `score` variable that you add to, a `maximum score` that you set by hand, and a table of rankings, and it ends the game with “end the story” or “end the story finally”. TADS 3 awards points through `Achievement` objects and ends with `finishGameMsg`. Chord’s `score` lines are nearest to the TADS approach, in that each award is a named thing, and TADS can be told to give it only once (`awardPointsOnce`). The maximum is computed for you, and an ending is one of three plain statements.

18.9 What you have so far

The story counts, keeps score and can be won or lost. What it cannot do is change by itself. Every line you have written so far runs because the player typed something. The last chapter of Part III is about things that happen whether the player acts or not.

19 Time

A story moves in **turns**. Each command the player types is one turn, and so far the world has held still between them. Ansel could stand in the Lodge for a week and dusk would never come.

Four commands are exceptions. `score`, `about`, `help` and `version` are questions put to the game and not acts in its world, and they take no turn. Nothing in this chapter runs after one of them, and none of them is counted.

This chapter gives the story a clock. It has four parts: things that happen every turn, timers that count a few named turns and then go off, a sequence that runs against the turn count of the whole game, and chapters that divide the game into named stretches.

19.1 Every turn

A clause whose head is `on every turn` runs at the end of each turn, after the player's command has been dealt with. It usually carries a condition, so that it runs only while something is true.

The cistern in the Undercroft has been frozen for thirty years. A lit lamp standing over it should thaw it. First the condition, with a name:

```
define condition lamplight-here: the hand lamp is lit and the hand
↪ lamp is here and the cistern is here
```

Then two clauses on the cistern.

```
create the cistern
  aka ice
  a thing, scenery
  in the Undercroft
  states: frozen, thawing, running
  score thaw worth 5

  on every turn while the cistern is thawing and lamplight-here
    change the cistern to running
    move the mend page to the Undercroft
    award thaw
    phrase cistern-gives
  end on

  on every turn while the cistern is frozen and lamplight-here
    change the cistern to thawing
    phrase cistern-drips
  end on
```

A thing may have several `on every turn` clauses, and they are tried in the order they are written. The order here is deliberate. On the first turn of lamplight the upper clause finds the cistern `frozen` and does nothing, and the lower clause changes it to `thawing`. On the next turn the upper clause runs. Written the other way round, the cistern would freeze, thaw and run within a single turn, because the first clause would set up the condition for the second.

```
> down
Undercroft
Low vaults on squat pillars, running off under the whole of the
hall. The air is cold and smells of stone. The steps go back up.

You can see a crate here.

The tower bell strikes once, and the sound goes through the stones
under your feet.

Where the lamplight falls on it, the ice in the cistern begins to
sweat.

> examine cistern
A stone cistern sunk in the floor, fed by a lead pipe from
somewhere above.

The ice has gone dark and wet, and stands a finger's width clear
of the stone all round.

The ice in the cistern breaks up all at once. A leaf of paper
rides up on the meltwater and slides out onto the flags.

The tower bell tolls.

You have risen to the rank of a Second-Year.
```

The second turn shows the order of events inside a turn. The player's command is answered first, and the cistern is still `thawing` when it is examined. Then the every-turn clause runs and the ice goes.

Three lines in that block are not the cistern's. The lamp will not light until the spark page has been read, and reading the page sets the bell going, so by the time lamplight reaches the Undercroft the bell is striking and tolling. The next section is about that. The last line is chapter 18's rank, earned by `award thaw`. If you lit the candles on the way, you had that rank already and the line does not appear.

The leaf of paper is the mend page, which has been offstage until now. It is declared like the other pages, with the states `faded, legible, learned`, a reading clause and a score of its own.

An `on every turn` clause may also be written on a region, where it runs while the player is in one of the region's rooms, and in the story header, where it runs wherever the player is.

19.2 Timers

A **timer** counts a small number of turns and then goes off. In Chord a timer does not count in numbers. You name its turns, and it steps through the names.

When Ansel reads the first working, the Hall takes three turns to notice.

```
define timer waking
  creak
    Somewhere above you, old timber takes a weight it had
    ↪ forgotten.
  sway
    A slow draught moves through the Hall, one way and then the
    ↪ other.
  strike
    The tower bell strikes once, and the sound goes through the
    ↪ stones
    under your feet.
end timer
```

Each line under the head is the name of one turn. The text indented beneath a name is printed on the turn the timer reaches it. A name with no text is a silent turn.

A timer does nothing until it is started, and it is started with a statement. The spark page starts this one, in the arm for its first reading.

```
      when legible
        change the spark page to learned
        change the story to waking
        start waking
        raise workings by 1
        award first-working
        phrase spark-learned
```

On the turns that follow, the timer speaks, whatever the player is doing.

```
> light lamp
You light the hand lamp.

Somewhere above you, old timber takes a weight it had forgotten.

> east
Cloister
A covered walk around a garth gone to nettles. The refectory is
east, the library door west, and the foot of the bell stair north.
Worn steps go down into the dark under the hall.

A slow draught moves through the Hall, one way and then the other.
```

After its last named turn a timer **expires**, and a clause can wait for that. This timer belongs to the story, so its clause goes in the story header.

```
story
  title: The Relighting
  ...
  when waking expires
    change the tower bell to tolling
  end when
```

The clause opens with `when`, the timer's name and `expires`, and closes with `end when`. The bell strikes once on the timer's last turn. On the turn after that the timer has expired, the bell is `tolling`, and the header's other clause begins.

```
on every turn while the tower bell is tolling
  select cycling
    phrase toll-near
  or
    phrase toll-far
  or
    phrase toll-count
  end select
  raise tolls by 1
end on
```

From that turn until something stops it, every turn ends with the bell.

```
> wait
Time passes...

The bell tolls again, and the rooks go up off the roof.
```

Nothing in the story can stop it yet. That is what the third working is for, and it arrives in chapter 24. Until then Lanternhall is a noisy place to test in, which is some encouragement to get there.

It also means that from here to the end of the book, any reply quoted from a game in which the spark page has been read ends with one more line. It is one of the timer's three on the first three turns after the reading, and one of the bell's three after that, until the bell is hushed. The blocks in this book print the line that came when the block was recorded. Yours will have a line in the same place, and which line it is depends on how many turns you have taken since you read the page.

19.3 More about timers

Five statements work a timer.

Statement	What it does
`start waking`	begins it, and does nothing if it is already running

Statement	What it does
`stop waking`	holds it where it is
`restart waking`	sends it back to its first turn, running
`reset waking`	puts it back to not started
`interrupt waking`	makes it expire now

A timer may belong to a thing. The head says so.

```
define timer fuse for the squib
  hiss
    The fuse hisses.
  spit
    The fuse spits.
  meanwhile
    phrase smoke
end timer
```

A timer that belongs to a thing is worked from that thing's block by its bare name, and its `when fuse expires` clause is written there too. Its named turns are spoken only where the thing is. A player who lights the fuse and walks into the next room does not read the hiss.

The `meanwhile` block holds statements that run on every turn the timer is running, beside the named turns. Add a chance to it, as in `meanwhile, one chance in 3`, and it runs only on some of them.

A condition can ask about a timer: `the squib's fuse has started` and `the squib's fuse has expired`.

Think of a timer as a countdown with a script. It is the right tool when something has been set going and a few turns later it happens. For something tied to the clock of the whole game, there is a different one.

19.4 A sequence

A **sequence** is a list of steps set against the turn count of the story. The Relighting gives Ansel one day.

```
define sequence the day
  at turn 50
    phrase light-turning
  30 turns later
    lose night-falls when the story is not awake
end sequence
```

The first step is anchored to a turn of the game with `at turn`. Each later step may be anchored the same way, or set a number of turns after the one before it with `turns later`. The statements under a step run when its turn comes. The opening of a story is not a turn, and neither is a command such as `score`, so turn fifty is the fiftieth command that takes a turn.

At turn fifty the player is warned.

```
> wait
Time passes...

The light has turned. The shadows in the Hall have begun to lean
east.
```

Thirty turns after that, if the Lantern is not burning, the day is over.

```
> wait
Time passes...

Dusk comes down on Lanternhall and finds the Lantern dark. The
blue flame in the porter's lodge goes yellow, and then goes out.

[The story has ended. RESTART, RESTORE or UNDO]
```

This is the third ending of The Relighting, and the first one a player can reach in the story as it stands. They need only wait.

A deadline changes a game more than any other single thing in this chapter. Eighty turns is generous for a walk that takes about fifty, and you should count your own walk before you choose a number. Count it again whenever you add a puzzle.

19.5 Chapters

A long story falls into parts, and it is useful to be able to say which part the player is in. The header line `use chapters` turns this on, and a `define chapters` block lists them.

```
define chapters
  morning — Morning
    begins when the game starts
  afternoon — Afternoon
    The Hall has noticed you.
    begins when the story becomes waking
  dusk — Dusk
    begins when the player visits the Lantern Gallery for the
    ↪ first time
end chapters
```

Each chapter has a name, a hyphen and a title. The name is the one word conditions will use. The title, and the optional line of description under it, are for the player. Every chapter says when it begins, and there are four ways to say it.

Trigger	Begins when
`the game starts`	at the beginning, and exactly one chapter must say this
`the story becomes waking`	a thing, or the story, reaches a state
`waking expires`	a timer goes off
`the player visits the Lodge for the first time`	the player first arrives in a room

In the terminal a new chapter prints nothing. The title and description are sent to the page the player sees, and chapter 37 shows what a browser does with them. What a chapter gives you here is three conditions: `during morning`, `before dusk` and `after morning`.

```
create the blue flame
  aka pilot flame, sill
  a thing, scenery
  in the Lodge

  Hobb's own lamp, which has burned yellow on that sill for as
  ↪ long as
  anyone has kept count. The flame is blue this morning, and does
  ↪ not
  flicker.

  after the player examining
    phrase flame-morning when during morning
    phrase flame-later when after morning
  end after
```

```
> examine blue flame
Hobb's own lamp, which has burned yellow on that sill for as long
as anyone has kept count. The flame is blue this morning, and does
not flicker.

It is early yet. The flame has the whole day to make up its mind.
```

Come back after reading the page and the flame says something else.

```
> examine blue flame
Hobb's own lamp, which has burned yellow on that sill for as long
as anyone has kept count. The flame is blue this morning, and does
not flicker.

It is taller than it was this morning, and it leans toward the
Hall.

The bell tolls. You have stopped counting.
```

A clause head may use the short form `during morning` where it would otherwise say `while`, as in `after the player examining during morning`.

Chapters and story states overlap, and The Relighting uses both on purpose. A state is something clauses change. A chapter begins by itself when its trigger comes true, and it never goes back. Use chapters for the large movements of the plot and states for the facts.

19.6 When something moves

One more clause waits for an event. A clause that opens `when Hobb moves` runs after that person has changed rooms. Nobody in Lanternhall moves by themselves yet, so it waits for chapter 26, where people do. It concerns people arriving. Picking a thing up and carrying it is not a move in this sense.

19.7 Build and try it

This is the largest change so far. Copy `relighting.story` from the companion folder `chapter-19` over your own file. It is the story with this chapter's changes made, and it holds the mend page, the fourth rank and the phrases, which the chapter did not print. To make the changes by hand instead, work from this list and take the rest from that file. Add to the header: `use chapters`, the `on every turn` clause and the `when waking expires` clause, and lengthen the rank ladder to four ranks ending at 35. At the top level add `define counter tolls`, the timer, the sequence, the chapters and the `lamplight-here` condition. Add the mend page, the cistern's score and two clauses, the `start waking` line, and the clause on the blue flame. There are eleven new phrases.

```
sharpee compose relighting.story --check
sharpee play .
```

Play through the reading of the page and down to the Undercroft with the lamp lit, and watch three things run at once: the timer counting, the cistern thawing, and the bell beginning. Read the mend page the thaw sets free, and ask for the score.

```
> score
You have scored 15 out of 35, earning you the rank of a Second-
Year.
```

No bell line follows it, though the bell is tolling: `score` takes no turn.

UNDER THE HOOD: WHO IS LISTENING · STORY IR `TIMERS`, `SEQUENCES`

The compiled IR lists the timer with its named turns as an ordered list of states, and the sequence with each step's timing. Look at the sequence's `narration` field: it says `broadcast`. Text printed by the story's own clauses and by a sequence reaches the player wherever they are. Text from a thing's every-turn clause is tied to the thing, which is why the timer that wakes the Hall belongs to the story and not to the bell.

FROM ANOTHER SYSTEM: SCHEDULED EVENTS

Inform 10 has "Every turn" rules and timed rules of the form "the bell tolls in three turns from now", and it has scenes, which begin and end on conditions. TADS 3 has `Daemon` and `Fuse` objects. Chord's `on every turn` is the every-turn rule and the daemon, and its chapters are a simpler cousin of scenes: they begin on a trigger, they come in a fixed order, and they do not end except by the next one beginning. The timer is the odd one out. It counts named turns where a fuse counts a number.

19.8 What you have so far

That is Part III. Lanternhall has states, clauses, conditions, a score, three endings and a clock. A player can read the first working, light the lamp, thaw the cistern, find three of the four pages, and run out of day.

What they read while doing it is still rough. Every refusal is one fixed sentence, the standard messages are the platform's and not this story's, and "Time passes…" is not how Lanternhall would put it. Part IV is about words.

PART IV. WORDS

20 Phrases

Every sentence a player reads in a Chord story is a **phrase**: a piece of text with a key. You have been writing them since chapter 12, in the one form that chapter needed, a key and its text in the `define phrases` block.

A phrase can be more than that. It can have several versions and a rule for choosing between them. It can keep its line breaks. It can leave a gap to be filled when it is printed, and it can be dropped into a room's description and appear there only when it is true. This chapter covers all four.

20.1 Two ways to declare a phrase

The block you know is a list.

```
define phrases en-US
  stair-sealed:
    The grille stands across the stair, and there is no way round
    ↪ it.
  over-the-parapet:
    You step up onto the parapet to see the court. The court comes
    ↪ up to
    meet you.
```

Each entry is one key and one paragraph. A story has one such block for each language it is written in, and `en-US` names the language. Most of your phrases will live here for the whole life of the story.

The second way declares one phrase in a block of its own.

```
define phrase gate-motto
  The craft needs a light to come home to.
end phrase
```

It opens with `define phrase` and the key, and closes with `end phrase`. For a plain phrase the two ways are equal. The block form exists because its head line has room for more.

A key belongs to one phrase. Declaring the same key twice, in either form or one of each, is an error.

20.2 Variants and strategies

Three things in The Relighting refuse with the same key, `no-such-word`: the lamp, the candles and the Lantern. A player without the first working will read that sentence several times. Give it three versions.

```
define phrase no-such-word, cycling
  You reach for the word and find only the place where it was.
or
  The shape of it is there, as a name is on the tip of the tongue.
  ↪ The
  word is not.
or
  Nothing comes. You would need to read it first.
end phrase
```

The versions are **variants**, separated by a line that says `or`. The word after the comma on the head line is the strategy, and the five strategies are the ones chapter 17 listed for `select`: `cycling`, `stopping`, `randomly`, `sticky` and `first-time`.

```
> light lamp
You reach for the word and find only the place where it was.

> light lamp
The shape of it is there, as a name is on the tip of the tongue.
The word is not.

> light lamp
Nothing comes. You would need to read it first.
```

No clause had to change. The three things still say `no-such-word`, and the phrase has become richer underneath them. Notice also that the cycle is shared. If the player tries the lamp, then the candles, then the lamp again, they get the first, second and third variants, because it is one phrase keeping one place.

This is the short way promised in chapter 17. When all you want is to vary some text, put the variants in the phrase. Keep `select` with a strategy for alternatives that do different things.

Remove the entry for `no-such-word` from the `define phrases` block when you add this one, or the key is declared twice.

20.3 The sayings of Hobb

The people of this country have a word they say about sunlight. They say it on opening the shutters and on parting, and they say it of anyone who is all cleverness and no warmth. The word is *gnusto*. Hobb says it the way other people clear their throats.

```
define phrase hobb-saying, cycling
  "Gnusto on the sill," says Hobb, without turning round. It is
  ↪ what
  people here say for good morning.
or
  "Gnusto keeps what the dark forgets," Hobb remarks, to the
  ↪ kettle.
or
  "All lamp and no gnusto, some people," says Hobb. He does not
  ↪ say
  which people.
end phrase
```

He says one each time Ansel comes into the Lodge, unless the bell is tolling, in which case nobody can hear him.

```
create the Lodge, a room
  in the Grounds
  north to the Gate Court

  room name:
    The Porter's Lodge

  The porter's lodge is one small room built into the thickness of
  ↪ the
  gate arch. A chair, a kettle and thirty years of newspapers have
  ↪ filled
  most of it. On the sill a lamp is burning with a steady blue
  ↪ flame, and
  an oak press hangs on the wall beside it. The door to the court
  ↪ is
  north.

  after the player entering
    phrase hobb-saying when the tower bell is not tolling
    phrase hobb-deafened when the tower bell is tolling
  end after
```

```
> south
The Porter's Lodge
The porter's lodge is one small room built into the thickness of
the gate arch. A chair, a kettle and thirty years of newspapers
have filled most of it. On the sill a lamp is burning with a
steady blue flame, and an oak press hangs on the wall beside it.
The door to the court is north.

You can see a kettle and Hobb here.

On the chair you see a seed cake and a cup.

"Gnusto keeps what the dark forgets," Hobb remarks, to the kettle.
```

20.4 Text that keeps its shape

Chord treats the text of a phrase as a paragraph. It joins your lines into one and lets the player's screen wrap them. That is right for prose and wrong for anything whose line breaks matter, such as an inscription.

The letters cut over the gate of Lanternhall are in three lines, centred.

```
define phrase gate-motto, verbatim
        THE CRAFT
     NEEDS A LIGHT
    TO COME HOME TO
end phrase
```

`verbatim` takes the place of a strategy. A verbatim phrase is printed line for line, and the indentation of each line relative to the others is kept. It has one variant and takes no `or`.

```
create the gate
  aka gates, arch, letters, motto
  a thing, scenery
  in the Lodge and the Gate Court

  The great gate of Lanternhall, shut for thirty years and opened
  ↪ this
  morning by one leaf. There is a line of cut letters over the
  ↪ arch.

  after the player examining
    phrase gate-motto
  end after
```

```
> examine gate
The great gate of Lanternhall, shut for thirty years and opened
this morning by one leaf. There is a line of cut letters over the
arch.

    THE CRAFT
 NEEDS A LIGHT
TO COME HOME TO
```

The line that sits furthest left in the file sits at the margin on the screen, and the others keep their distance from it.

20.5 Markers

A **marker** is a pair of braces in the text of a phrase, with something inside that is filled in when the phrase is printed.

The simplest marker is `{br}`, which is a line break inside an ordinary paragraph.

The most useful markers take a value handed over by the clause. The `phrase` statement can pass one, with the word `with`, a name for it, an equals sign and the value.

```
create the tower bell
  aka bell
  a thing, scenery
  in the Bell Stair
  states: unrung, tolling, silent

  You cannot see the bell from here, only the dark it hangs in.

  after the player examining
    phrase bell-count with count = tolls when the tower bell is
    ↪ tolling
  end after
```

```
define phrases en-US
  bell-count:
    By your count that makes {verbatim:count}, and it is not
    ↪ tired.
```

The statement passes the counter `tolls` under the name `count`, and the marker `{verbatim:count}` prints it as it stands.

```
> examine bell
You cannot see the bell from here, only the dark it hangs in.

By your count that makes 3, and it is not tired.

The tower bell tolls.
```

When the value is a thing, the marker can ask for its name in the form the sentence needs.

Marker	Prints
`{the thing}`	the lamp
`{a thing}`	a lamp
`{capitalize the thing}`	The lamp
`{verbatim:count}`	a number or a piece of text, unchanged

Here `thing` and `count` are the names given after `with`. The statement that goes with the first three would be `phrase named with thing = the hand lamp`.

Markers of this kind matter most in a phrase that several things share, and you will use them in earnest in chapter 23, where one refusal has to name whichever thing the player typed.

One caution. A marker with a plain lower-case word and nothing else, such as `{count}`, is not read as a value. It is read as the key of another phrase, which is the subject of the next section, and the compiler will tell you that no such phrase exists.

20.6 A phrase inside a description

A room's description is fixed text, and by chapter 19 that had become a problem. The Bell Stair's description says that an iron grille stands across the third step. One day the grille will be gone, and the description will go on saying so.

A marker in a room's description may name a phrase. The phrase's text is set into the description at that point.

```
create the Bell Stair, a room
  in the Hall
  up to the Lantern Gallery
  up is blocked while the iron grille is standing: stair-sealed

  The foot of the tower. A stair winds up around the well where
  ↪ the bell
  rope hangs. The cloister is south. {grille-across}
```

And the phrase can carry a condition on its head line.

```
define phrase grille-across while the iron grille is standing
  An iron grille stands across its third step.
end phrase
```

While the grille is standing, the room reads as it always did.

```
> north
Bell Stair
The foot of the tower. A stair winds up around the well where the
bell rope hangs. The cloister is south. An iron grille stands
across its third step.
```

When the grille is lifted, the sentence is simply absent, and nothing in the room's block had to know.

A phrase set into a description may have variants and a strategy as well, and then the description changes a little each time the player looks.

Three things to know about this form.

It works in the description of a room. A marker in the description of a thing is not filled in. For a thing, the tool is the `detail` phrase of the next chapter.

The `while` on a phrase's head governs its appearance in a description. It does not stop a clause from printing the phrase with a `phrase` statement. To make a statement conditional, use the `when` suffix, as the Lodge's clause does.

Put the marker at the end of a paragraph when you can. A marker in the middle leaves its two spaces behind when the phrase is absent.

20.7 Build and try it

Copy `relighting.story` from the companion folder `chapter-20` over your own file. It is the story with this chapter's changes made. To make the changes by hand instead, work from this list and take the rest from that file. This is the first chapter of Part IV, so raise `story-version` to `0.4.0`. Add the four `define phrase` blocks near the top of the file and remove `no-such-word` from the `define phrases` block. Add the clauses to the Lodge, the gate and the tower bell, change the gate's description, put the marker in the Bell Stair's description, and add the two new entries `bell-count` and `hobb-deafened`.

```
sharpee compose relighting.story --check
sharpee play .
```

Walk in and out of the Lodge three times to hear all of Hobb's sayings. Then read the page, wait for the bell, and go back.

```
> south
The Porter's Lodge
The porter's lodge is one small room built into the thickness of
the gate arch. A chair, a kettle and thirty years of newspapers
have filled most of it. On the sill a lamp is burning with a
steady blue flame, and an oak press hangs on the wall beside it.
The door to the court is north.

You can see a kettle and Hobb here.

On the chair you see a seed cake and a cup.

Hobb has his hands over his ears. He says something, and the bell
says something louder.

The tower bell tolls.
```

UNDER THE HOOD: ONE TABLE OF PHRASES · STORY IR PHRASES

Both ways of declaring a phrase end up in one table in the compiled IR, keyed by language and then by phrase key. Each entry has a `strategy`, which is empty for a plain phrase, and a list of `variants`, each with its text and the markers found in it. Descriptions are in the same table under keys the compiler makes up, such as `spark-page.description`, which is why a description and a phrase can be treated so alike.

FROM ANOTHER SYSTEM: TEXT SUBSTITUTIONS

Inform 10 puts substitutions in square brackets inside the text, and "[the noun]" and "[The noun]" do what `{the thing}` and `{capitalize the thing}` do here. TADS 3 uses "{the dobj/him}" style parameters in message strings, and `<<expression>>` to embed a value. The idea is common to all three. Chord differs in keeping its text out of the code entirely: a clause names a key, and the words are somewhere else, where a writer or a translator can work on them without reading a single clause.

20.8 What you have so far

Phrases can vary, keep their shape, take values and appear in a room's description only while they are true. All of that is about text the story prints when something happens. The next chapter is about the words a thing carries around with it.

21 A thing’s own words

The phrases of the last chapter belong to the story. Any clause may print any of them. Some words belong to one thing: the sentence that says the lamp is burning, the line that tells the player Hobb is in the room. Those are written in the thing’s own block, with a `phrase` line that is not a statement and does not sit inside a clause.

There are a handful of these, each with a reserved key. This chapter covers the three you will use most, `detail`, `present` and `disappeared`, and then the general rule they are examples of.

21.1 Detail: a sentence that depends

A description is fixed. The hand lamp’s description says that its wick is dry, and it says so whether the lamp is burning or not. Chapter 17 handled the cistern with an `after` clause and a `select` block, which is the right tool when a thing has several states and something to say in each. For one added sentence there is a lighter form.

```
create the hand lamp
  a thing, light-source, starts unlit
  in the Lodge

  A tin hand lamp with a wire handle. Its wick is dry and its
  ↪ glass is
  clean, which is Hobb all over.

  on the player lighting
    the spark page must be learned: no-such-word
  end on

  phrase detail while the hand lamp is lit:
    It is burning with a small flame, and the flame is blue.
```

The line is `phrase detail`, the word `while`, a condition and a colon, with the text indented beneath. While the condition holds, the text is added to the end of the description.

```
> examine lamp
A tin hand lamp with a wire handle. Its wick is dry and its glass
is clean, which is Hobb all over.
```

Read the spark page, light the lamp and look again. The Hall's waking timer is running by then, and it has its say at the end of each turn.

```
> light lamp
You light the hand lamp.

Somewhere above you, old timber takes a weight it had forgotten.

> examine lamp
A tin hand lamp with a wire handle. Its wick is dry and its glass
is clean, which is Hobb all over. It is burning with a small
flame, and the flame is blue.

A slow draught moves through the Hall, one way and then the other.
```

The sentence joins the paragraph. It is not printed as a line of its own, as a clause's phrase would be, and that is the difference between the two. Detail is part of what the thing looks like.

Three rules go with it. A `detail` phrase must have a `while`, because a sentence that is always true belongs in the description. It has one text and no variants. And a thing may have as many `detail` phrases as it needs.

The candles show why you would want two. The old description said that nobody had put a flame to them since the last supper of the last term, and that stops being true the moment they are lit. So the description is cut back to what is always so, and each condition gets its sentence.

```
create the candles
  aka prickets
  a thing, scenery, plural, light-source, starts unlit
  in the Refectory

  Stubs of tallow in iron prickets down the middle of each table.

  phrase detail while the candles is not lit:
    Nobody has put a flame to them since the last supper of the
    ↪ last
    term.

  phrase detail while the candles is lit:
    They are all alight, every one, and the flames stand very
    ↪ straight.
```

```
> examine candles
Stubs of tallow in iron prickets down the middle of each table.
Nobody has put a flame to them since the last supper of the last
term.

The bell tolls again, and the rooks go up off the roof.

> light candles
You light the candles.

The flame runs down the length of the table, and every cold wick
takes it. In the candlelight there is a leaf of paper under the
middle pricket that the daylight did not show.

The bell tolls. You have stopped counting.

You have risen to the rank of a Second-Year.

> examine candles
Stubs of tallow in iron prickets down the middle of each table.
They are all alight, every one, and the flames stand very
straight.

The tower bell tolls.
```

Look at the condition: `the candles is lit`. The candles are plural, and English wants “are”. Chord does not. A condition’s verb is always `is`, whatever the thing. Written as `the candles are lit`, the line is refused twice, because the word is neither a verb a condition may use nor the name of a condition.

```
relighting.story:405:35 error [parse.predicate] Unknown predicate
`are` — expected is, is a, is in, has, holds, wears, feels, knows,
or can see/reach.
relighting.story:405:35 error [analysis.unknown-condition] `are`
is not a declared condition or story state.
compose: relighting.story failed the load-time gates (2 error(s))
```

The lens takes the same treatment. Its crack moves out of the description and into a detail phrase, and a second phrase stands ready for the day it is mended.

```
create the lens
  a thing
  in the crate
  states: cracked, sound

  A disc of glass as wide as a cartwheel, ground in rings.

  phrase detail while the lens is cracked:
    A crack runs across it from edge to edge.

  phrase detail while the lens is sound:
    There is no flaw in it anywhere, and you looked.
```

21.2 Detail or a clause?

You now have two ways to make an examination depend on state, and they are not rivals.

Use `detail` when the thing has a description and the state adds a sentence to it. The result reads as one paragraph, and the condition can be anything, including a platform state such as `lit`.

Use an `after the player examining` clause with `select on` when the thing moves through several declared states and each has its own text, as the cistern does, or when examining should do something as well as say something. The grille's three memories are a clause, because they count.

When you find yourself writing three or four `detail` phrases whose conditions are the states of one list, that is a `select on`.

21.3 Present: how a thing is announced

When the player enters a room or looks around, the story lists what is there: "You can see a kettle, a hand lamp, and Hobb here." For most things that is enough. For a person, or for anything the eye would go to first, it is flat.

A `phrase present` line gives a thing its own sentence in the room.

```
create Hobb
  a person, proper
  in the Lodge

  The porter of Lanternhall, who was old when you were a student
  ↪ and has
  not troubled to change since.

  phrase present:
    Hobb is in the doorway with his back to you, watching the
    ↪ court as
    if it might be up to something.
```

```
The Porter's Lodge
The porter's lodge is one small room built into the thickness of
the gate arch. A chair, a kettle and thirty years of newspapers
have filled most of it. On the sill a lamp is burning with a
steady blue flame, and an oak press hangs on the wall beside it.
The door to the court is north. Hobb is in the doorway with his
back to you, watching the court as if it might be up to something.

You can see a kettle, a hand lamp, and Hobb here.

On the chair you see a seed cake and a cup.
```

The sentence follows the room's description wherever Hobb is. If a later chapter walks him to the Cloister, the sentence goes with him, and the Lodge's description has nothing to take back. That is why the Lodge was never written to mention him.

A `present` phrase may carry a `while`, and then the thing announces itself only while the condition holds. A thing has one `present` phrase. Where `detail` may be repeated, a second `present` is refused as a duplicate.

The thing is still named in the list beneath. The sentence is an addition to the room and does not take the thing out of the list.

21.4 Disappeared: a parting line

The third reserved key speaks when a thing is taken out of the story with `remove` in front of the player.

```
create the cat
  a thing
  in the Yard

  A cat.

  phrase disappeared:
    The cat is gone, and has left a look behind it.
```

The line is printed on the turn the `remove` runs. It saves you from writing a `phrase` statement beside every `remove`, and it keeps the cat's exit with the cat.

Two more keys, `entered` and `exited`, are reserved for a thing that comes into the player's room or leaves it. They are for things and people that move about, and this book takes them up in chapter 26 with the people who do the moving.

21.5 The general rule: a thing may answer for itself

The reserved keys are particular cases of something broader. A `phrase` line in a thing's block is that thing's own version of a piece of text.

You will see the broader use in chapter 24. A trait there declares a phrase that every thing with the trait prints when it is mended. One thing with the trait wants a sentence of its own, and it gets one by writing a `phrase` line with the same key in its block. The trait says what happens. The thing says how it reads.

That division is the one this book keeps returning to. What happens is a clause, and it lives with whatever causes it. How it reads is a phrase, and it lives with whatever it describes.

21.6 Build and try it

Copy `relighting.story` from the companion folder `chapter-21` over your own file. It is the story with this chapter's changes made. To make the changes by hand instead, work from this list and take the rest from that file: add the `detail` phrases to the hand lamp, the candles and the lens, cut the candles' and the lens's descriptions back to what is always true, and add the `present` phrase to Hobb. No new entries go in the `define phrases` block.

```
sharpee compose relighting.story --check
sharpee play .
```

Take the lit lamp down to the Undercroft and open the crate.

```
> examine lens
A disc of glass as wide as a cartwheel, ground in rings. A crack
runs across it from edge to edge.

The bell tolls again, and the rooks go up off the roof.
```

UNDER THE HOOD: OWNER AND KEY · STORY IR PHRASES

In the phrase table of the compiled IR, a thing's own phrases are stored under the thing's id, a dot and the key: `hobb.present`, `hand-lamp.detail`, and `candles.detail` with `candles.detail.2` for the second. The dot is why a key of your own may never contain one. The conditions are kept beside them, and the platform asks each one when it builds a description or lists a room.

FROM ANOTHER SYSTEM: LISTING AND DESCRIBING

Inform 10 handles the same needs with an "initial appearance" for a thing not yet handled, and with "writing a paragraph about" rules. TADS 3 gives objects `specialDesc` for a sentence in the room description and `initSpecialDesc` for a thing that has not been moved yet. Chord's `present` is close to `specialDesc`, with the difference that it is a phrase with an optional condition and no code, and `detail` has no direct equal in either system: both would do it with a conditional inside the description text.

21.7 What you have so far

Things in Lanternhall now describe the condition they are in, and the porter is a presence in his lodge and not an item in a list. Everything the story itself says is in your words. What remains is everything the platform says: “Taken”, “Time passes”, and a line about a grue that has no business in this country.

22 Changing the standard messages

Most of what a player reads in a small story was not written by its author. “Taken.” “Time passes…” “You open the oak press.” Those sentences come with the standard actions, and they are plain on purpose, so that they fit anywhere. This chapter is about making them fit here.

It also covers the phrasebook, which is the tool for the opposite problem: text of your own that should be said one way at one time and another way later.

22.1 Finding a message

Every standard message has an **alias**, a hyphenated name made of the action and the message. The `sharpee` command will list them all.

```
sharpee messages
```

The list is long, 885 messages at this edition, and it is printed as data for a program to read: one line that gives the count, and then the whole list as a single very long line. Each entry in that line looks like this.

```
{"id":"if.action.taking.taken","alias":"taking-
taken","template":"Taken."}
```

The `alias` is what you will write. The `template` is the text the platform prints, and reading it tells you whether you have found the right message. Appendix F is this list, sorted by action, and it is the place to look things up. To find a message without the appendix, send the list to a file, open the file in an editor, and use the editor’s search for a few words of what the game printed. A search that prints whole matching lines will print the entire list, because the list is one line.

22.2 Overriding one message

Walk into the Undercroft without a light and the platform says this.

```
> down
It is pitch dark. You are likely to be eaten by a grue.
```

It is a famous sentence, and it belongs to another country. Nothing lives in the dark under Lanternhall. Searching the list for “grue” finds the alias.

```
{"id":"if.action.going.too_dark","alias":"going-too-
dark","template":"It is pitch dark. {You} {be} likely to be eaten
by a grue."}
```

An override is a block at the top level of the file.

```
override message going-too-dark
  You are at the foot of the steps, in the dark, and that is all
  ↪ you
  can say for certain. You were a student here, and you remember
  ↪ what
  the Undercroft does to shins.
end override
```

It opens with `override message` and the alias, holds the new text, and closes with `end override`.

```
> down
You are at the foot of the steps, in the dark, and that is all you
can say for certain. You were a student here, and you remember
what the Undercroft does to shins.
```

If you mistype an alias, the compiler knows the list as well as you do.

```
probe.story:7:1 error [analysis.unknown-message-alias] `taking-
takn` is not a standard-action message alias — did you mean
`taking-taken`?.
```

22.3 An override is a phrase

The body of an override is the body of a `define phrase` block, and the head line takes the same additions. So a standard message may have variants and a strategy.

```
override message waiting-time-passes, cycling
  You wait. The Hall has had thirty years of practice and is
  ↪ better at
  it than you are.
or
  You wait. Somewhere a shutter knocks, twice, and thinks better
  ↪ of it.
or
  You wait.
end override
```

```
> wait
You wait. The Hall has had thirty years of practice and is better
at it than you are.

> wait
You wait. Somewhere a shutter knocks, twice, and thinks better of
it.

> wait
You wait.
```

Waiting is a good message to spend this on. A player in a game with a clock will wait often, and "Time passes…" thirty times is thirty reminders that they are playing a program.

22.4 Several at once

When you are replacing a number of messages with one plain text each, a single block holds them all.

```
override messages en-US
  taking-taken:
    You take it.
  looking-room-dark:
    It is dark, and the dark here is old and has no interest in
    ↪ you.
```

This is the list form, the counterpart of `define phrases`. It names its language, and each entry is an alias, a colon and one paragraph. An entry here cannot have variants. A message that needs them gets a block of its own.

```
> take lamp
You take it.
```

22.5 The markers in a template

Many templates have markers in them.

```
{"id":"if.action.lighting.lit","alias":"lighting-
lit","template":"{You} {light} {the target}."}
{"id":"if.action.taking.taken_from","alias":"taking-taken-
from","template":"{You} {take} {the item} from {the container}."}
```

`{You}` is the player, capitalised because it opens the sentence. `{light}` and `{take}` are verbs, written so that they will agree with whoever is acting. `{the target}` and `{the item}` are the things in the command, by the names the action gives them, and they are the same kind of marker as `{the thing}` in chapter 20.

Your override may use the markers its template uses.

```
override message taking-taken-from
  {You} {take} {the item} out of {the container}.
end override
```

Use them whenever the message is about a thing. An override that says "You take the key out of the press" in so many words will say it of the lamp as well. And keep to the markers the template shows. Each message is given the values it needs and no others, so a marker borrowed from a different message has nothing to fill it.

22.6 How much to override

A new author either leaves every standard message alone or sets out to replace all 885. Neither is right.

Replace the messages your players will see most: taking, dropping, waiting, the two darkness messages, the refusal for a direction with no exit. Replace any message whose tone is wrong for your story, as the grue was. Then stop, play the game through, and replace what still jars. A story usually settles at a few dozen overrides, and the rest of the list goes unread because the player never does the thing that prints it.

Before you override a message to change what one thing says, ask whether that thing should say it. The kettle's refusal is a clause on the kettle, and `taking-taken` was left alone for it. An override changes a sentence everywhere. A clause changes it for one thing.

22.7 Phrasebooks

Now the opposite problem. Two of the story's own refusals should change once the Hall begins to wake. While it sleeps, the grille "does not so much as rattle". Once it is waking, the iron should feel alive. The same goes for Hobb and his kettle.

You could put a `when` suffix on every `refuse`, except that `refuse` does not take one. You could split each key in two and choose between them in every clause that uses them. A **phrasebook** does it without touching a clause.

A phrasebook is a named set of phrases with one condition on the whole set.

```
define phrasebook hall-waking while the story is not asleep
  kettle-is-hobbs:
    "Still mine," says Hobb, with his eye on the ceiling.
  grille-fixed:
    It hums under your hand, very faintly, like a rail with a
    ↪ train a
    long way off. It does not move.
end phrasebook

define phrasebook hall-asleep
  kettle-is-hobbs:
    Hobb does not look round. "That's mine," he says.
  grille-fixed:
    It does not so much as rattle.
end phrasebook
```

The two books hold the same keys. When a clause asks for `grille-fixed`, the story looks through the phrasebooks in the order they appear in the file and takes the entry from the first book whose condition holds. A book with no `while` always holds, so it is the book of last resort.

```
> push grille
It does not so much as rattle.
```

After the spark page has been read:

```
> push grille
It hums under your hand, very faintly, like a rail with a train a
long way off. It does not move.

The tower bell strikes once, and the sound goes through the stones
under your feet.
```

The grille's clause still says `refuse grille-fixed`, as it has since chapter 14.

Three rules keep phrasebooks working.

Order decides. Put the conditional books first and the book with no condition last. If `hall-asleep` stood first it would always be chosen, since it always holds, and the waking book would never be reached.

A key lives in the books or outside them. A key declared by `define phrases` or `define phrase` always wins over a phrasebook. When you move a key into phrasebooks, take its entry out of the `define phrases` block.

Books are for your keys. An entry in a phrasebook does not change a standard message. Those are changed by `override message`, and an override has no condition.

An entry may have variants and a strategy, written as in the list form with the strategy after the key: `kettle-is-hobbs, cycling:` and then the variants with `or` between them. What it may not have is a `while` of its own. The condition belongs to the book.

22.8 When to reach for a phrasebook

A phrasebook is the right tool when a mood shifts and several pieces of text should shift with it: before and after the storm, sober and drunk, the Hall asleep and the Hall awake. One line turns the whole set over.

It is the wrong tool for a single thing in a single state, which is a `detail` phrase or a `select on`. And it is easy to overuse. Each book you add is another place a reader has to look to find out what a key says. Two or three books is a design. Ten is a puzzle.

22.9 Phrases in a file of their own

A story's text can outgrow one file. Phrasebooks may be kept in a second file and brought in with one line.

```
import "books"
```

The name in quotes is a file beside the story, `books.chord`. The extension is left off. That file holds `define phrasebook` blocks and comments and nothing else, and it is not a story by itself: it has no header and cannot be built alone. Its books take their place in the order at the point where the `import` line stands.

A project has one `.story` file. Everything else it imports is a `.chord` file, and you will meet a second use for one in chapter 25.

22.10 Build and try it

Copy `relighting.story` from the companion folder `chapter-22` over your own file. It is the story with this chapter's changes made. To make the changes by hand instead, work from this list and take the rest from that file: add the override for `going-too-dark`, the override for `waiting-time-passes`, the `override messages` block and the two phrasebooks, and remove `kettle-is-hobbs` and `grille-fixed` from the `define phrases` block.

```
sharpee compose relighting.story --check
sharpee play .
```

Go down into the Undercroft without the lamp, and look.

```
> look
It is dark, and the dark here is old and has no interest in you.
```

UNDER THE HOOD: IDS AND ALIASES · `IF.ACTION.GOING.TOO_DARK`

Each entry of `sharpee messages` has an `id` beside the alias. The id is the platform's own name for the message, and it shows how the platform is arranged: `if.action`, then the action, then the message. The alias is the same name made safe for Chord, where a key may not contain a dot. In the compiled IR your overrides are stored under `messageOverrides` by alias, and your phrasebooks under `phrasebooks`, each with its condition and its entries.

FROM ANOTHER SYSTEM: LIBRARY MESSAGES

Inform 10 keeps its standard replies as lettered "responses" attached to rules, and you change one with a sentence such as: The can't take yourself rule response (A) is "…". TADS 3 keeps them in message objects, `playerActionMessages` and its relatives, which you modify. Chord's aliases are the same idea with names in place of rule letters. Neither system has a construct quite like the phrasebook. In both you would test a condition inside the message itself.

22.11 What you have so far

Lanternhall speaks with one voice. Its darkness is its own, waiting has three things to say, and the Hall sounds different once it has begun to wake. Every command a player can type is still one the platform supplied, though, and this is a story about four words that are not in any standard list. The next chapter adds verbs.

23 New verbs

The craft of Lanternhall calls its spells workings. A working is a single spoken word: spark, mend, hush, unbind. Ansel learns one by reading its page, and then says it to a thing. None of those four words is a command the platform knows.

This chapter adds them. A new verb in Chord is a new action, and you declare one with `define action`. By the end of the chapter all four workings can be typed, each asks for its page, and each then finds that nothing answers it. That last part is not a failure of the chapter. It is the most important idea in it.

23.1 Declaring an action

```
define action mending
  grammar
    mend the target
  the target must be reachable
  the mend page must be learned: no-such-word
  otherwise refuse nothing-to-mend
```

The head line names the action. An action's name is a gerund, the same "-ing" word that will later appear in clause heads as `on the player mending`. The block has no `end` line. It runs until the indentation stops.

The four lines beneath are four different kinds of line, and each gets a section.

23.2 The grammar

The `grammar` block lists what the player may type, one pattern to a line. You have read patterns before, in the passage of chapter 12. They have four devices.

Written	Means
`mend`	a word the player types
`the target`	a **slot**: a place where the player names a thing
`mend or fix`	either word
`[the]`	a word that may be left out

A slot is `the` followed by a name you choose. `target` is the usual name for the one thing a verb is done to. The name matters only inside the action, where the other lines use it to talk about whatever the player named.

An action may have several patterns, and a pattern may have no slot at all.

```
define action sparking
  grammar
    spark the target
    spark
```

Now `spark lamp` and plain `spark` are both understood. The second needs a line to say what happens when the thing is missing.

```
  refuse without target: working-wants-a-thing
```

```
> spark
A working is said to something. You say it to the air, and the air
has heard it all before.
```

One limit applies to every pattern you write. It may not be a command the standard grammar already has.

```
probe.story:37:1 error [analysis.repeats-standard-grammar]
`poke :target` is already the standard `touching` grammar, so this
action would take that command from it. To give a thing its own
response, write `on the player touching` in that thing's block or
its trait's. To add new wording, use `extend action touching`. To
replace the standard action, first remove its shape with `remove
from action touching`.
```

The message gives the three ways out, and chapter 25 is about the second and third. Before you define an action, look in Appendix C for the verb. The standard grammar is larger than most authors expect.

23.3 What must be true of the thing named

```
  the target must be reachable
```

A line of this form, with no colon and no phrase, says how close the named thing has to be. There are three words for it.

Word	The thing must be
`visible`	in view
`reachable`	near enough to touch
`held`	in the player's hands

When the thing is not there, the platform says so in its own words. Hushing uses `visible`, because a bell at the top of a tower can be spoken to and cannot be touched.

```
> hush bell
You can't see any bell here.
```

That was typed in the Library. In the Bell Stair, the bell is in view.

23.4 What must be true of the player

```
  the mend page must be learned: no-such-word
```

This is the `must` line of chapter 15, and it means what it meant there. It is on the action, and not on any thing, because it concerns the person speaking. Ansel cannot say a word that has not been read, whatever it is said to.

```
> mend lamp
The shape of it is there, as a name is on the tip of the tongue.
The word is not.
```

An action may also carry `refuse when` lines, and statements of its own, as a later section shows. All of its refusals and requirements stand above its first statement.

23.5 What an action may not ask

Here is the rule that shapes everything else. An action never asks which thing was named.

You cannot write, in the mending action, that if the target is the lens then the crack closes. Chord has no way to say it. If you try, with a line such as `refuse when the target is the kettle`, the compiler refuses it by name, as `analysis.slot-identity-test`.

The reason is the habit this book has pressed since chapter 14. What happens to the lens belongs in the lens's block. If the mending action listed the things it could mend, then adding a mendable thing to the story would mean editing the action, and the facts about the lens would be in two places.

So an action knows three things: what the player may type, what must be true of the player, and what to say when nothing answers. Everything else belongs to the thing.

23.6 When nothing answers

```
  otherwise refuse nothing-to-mend
```

When the player names a thing, the action asks that thing whether it has an answer. If it has none, this line speaks.

At the end of this chapter no thing in Lanternhall has an answer to any of the four workings. So once the page has been read, every one of them misses.

```
> spark lamp
There is nothing in that for a flame to stand on.

Somewhere above you, old timber takes a weight it had forgotten.

> spark grimoire
There is nothing in that for a flame to stand on.

A slow draught moves through the Hall, one way and then the other.
```

The first of those is plainly wrong, and it is wrong in an instructive way. The action is complete. The lamp has not yet been told that it answers to sparking. Telling it is the whole of the next chapter.

A miss can name the thing the player typed. The slot's name works as a marker.

```
define phrases en-US
  nothing-to-tune:
    {capitalize the target} is as well tuned as it will ever be.
```

23.7 The four workings

Here are the other three.

```
define action sparking
  grammar
    spark the target
    spark
  refuse without target: working-wants-a-thing
  the target must be reachable
  the spark page must be learned: no-such-word
  otherwise refuse nothing-to-spark

define action hushing
  grammar
    hush the target
    hush
  refuse without target: working-wants-a-thing
  the target must be visible
  the hush page must be learned: no-such-word
  otherwise refuse nothing-to-hush

define action unbinding
  grammar
    unbind the target
    unbind
  refuse without target: working-wants-a-thing
  the target must be reachable
  the unbind page must be learned: no-such-word
  otherwise refuse nothing-to-unbind
```

Mending takes the same two extra lines. The four blocks are nearly identical, and Chord offers no way to write one and stamp out four. That is a real cost, and the answer for now is that four short blocks are easy to read and easy to keep in step.

23.8 A verb that only answers

Not every verb is done to a thing. Some are simply said. There is a fifth word in this country that every child knows and that nobody at Lanternhall ever managed to cast.

```
define action blessing
  grammar
    gnusto
    say gnusto
    gnusto the target
  refuse when the tower bell is tolling: gnusto-drowned
  phrase gnusto-said
```

This action has a body. After its refusals come ordinary statements, and they run whenever the action succeeds. There is no `otherwise refuse`, because nothing needs to answer. The action answers for itself.

```
> gnusto
"Gnusto," you say, as everybody does, and the morning is exactly
as bright as it was. It is a word about sunlight. The scholars
here always said it had been a working once, somewhere over the
water, and could never say what it did.
```

Use a body for a verb that is about the player or the story: praying, singing, counting the money. An action's body may do anything a clause can do. It may change the story's state, raise a counter or award a score, and an action may declare a `score` line of its own. It still may not ask what was named.

The third pattern, `gnusto the target`, lets a player try the word on a thing. Nothing answers, and the body runs all the same.

23.9 Verbs with a subject

Some verbs take a subject and not a thing: remember the fire, think about home. The subject may be something that is not in the room, or not a thing at all. A slot can be declared a **topic**, and then the action answers by subject, with `about` lines.

```
define action remembering
  grammar
    remember the subject
    recall the subject
  the subject is a topic
  about "dimming":
    phrase remember-dimming
  about "gnusto":
    phrase remember-gnusto
  about Hobb:
    phrase remember-hobb
  otherwise refuse remember-nothing
```

An `about` line names a subject and holds the statements to run for it. A subject in quotes is matched against the words the player typed. A subject without quotes is a thing or a person in the story, matched by any of its names.

```
> remember dimming
You remember it as a night when every lamp in the Hall went out at
once, and a morning when nobody could make one light.

> remember the dimming
You remember it as a night when every lamp in the Hall went out at
once, and a morning when nobody could make one light.

> recall hobb
He confiscated a kettle from you in your second week. You have a
feeling it is the same kettle.

> remember breakfast
Nothing comes. Thirty years is a long time.
```

The article is dropped before matching, so `"dimming"` answers both of the first two commands. Write each subject once, without “the”. A quoted line may list several spellings with commas between them, as in `about "dimming", "night"`.

This does not break the rule about asking what was named. The rule is about things in the world, which have blocks of their own to answer in. A subject has no block. The `about` lines are where its answer lives.

You will meet `about` lines again in chapter 27, where they are what a person says when asked about something. An action with a topic is the same machinery pointed at the player's own head.

23.10 Other kinds of slot

Two more lines describe a slot.

`the words takes the rest of the line` makes a slot swallow everything the player types after the verb. It is for writing, saying and naming.

```
define action scrawling
  grammar
    scrawl the words
  the words takes the rest of the line
  phrase scrawled
```

The text is available to a phrase as `{verbatim:words}`.

```
> scrawl hello there world
[scrawled: hello there world]
```

`the tool is an instrument` marks a second slot as the thing the action is done with, as in a pattern `tune the target with the tool`. The standard grammar uses it for keys, weapons and tools.

23.11 A verb that comes and goes

An action's head may carry a condition. The verb then exists only while the condition holds.

```
define action shouting while the tower bell is tolling
  grammar
    shout
  phrase shout-lost
```

Before the bell begins, the story does not know the word.

```
> shout
I don't understand that.
```

Once it is tolling:

```
> shout
You shout. The bell tolls through the middle of it.
```

This is stronger than a refusal. A refused verb tells the player that the command exists. A verb that is not understood tells them nothing. Use it when the existence of the command would itself give something away, and otherwise prefer a `refuse when` with a sentence that helps. The Relighting does not use this action. It is here as an example.

23.12 Why a thing cannot simply answer

You may already have tried the obvious thing: an `on the player mending` clause in the lens's block, like the `on the player reading` clause on each page.

```
probe.story:19:17 error [analysis.story-action-on-clause] `on the
player thumping` in `drum`'s block would never run: `thumping` is
this story's own action, and it runs a thing's `after` clauses and
the `on` clauses of its traits, never a thing's own `on` clause.
To react once the action succeeds, write `after the player
thumping`. To refuse or answer in the action's place, move the
clause into a trait and give `drum` that trait.
```

For a standard action, a thing's own `on` clause is how it answers. For an action you define, the answer comes from a **trait**. The message says what to do, and the next chapter does it.

23.13 Build and try it

Copy `relighting.story` from the companion folder `chapter-23` over your own file. It is the story with this chapter's changes made, and it holds the phrases, which the chapter did not print in full. To make the changes by hand instead, work from this list and take the rest from that file: add the four working actions, the `blessing` action and the `remembering` action near the top of the file, and add their eleven phrases.

```
sharpee compose relighting.story --check
sharpee play .
```

Try each working before and after its page. The lamp still lights with the standard `light lamp`, as it has since chapter 15. Spark is a word now, and has nothing to work on.

UNDER THE HOOD: AN ACTION IS DATA · STORY IR `ACTIONS`

In the compiled IR the sparking action is an entry of plain data. It has a `name`, and then its `patterns`, each a run of words and slots; its `constraints`, which are the `must be reachable` lines; its `musts`, each a condition and a phrase key; its `refusals`, which are the `refuse` lines; `otherwise`, which holds the phrase key of the `otherwise refuse` line; `scores`, which is empty for this action; and its `body`. A last field, `span`, records where in the file the action was written. There is no code in it. The platform has one piece of machinery that runs any action described this way, and that machinery is what looks for an answer on the thing and falls back to your `otherwise refuse` line.

FROM ANOTHER SYSTEM: NEW ACTIONS

In Inform 10 you write "Mending is an action applying to one touchable thing" and "Understand 'mend [something]' as mending", and then rules say what it does, including rules that name particular things. In TADS 3 you define the action with `DefineTAction` and a `VerbRule`, and each object answers in its `dobjFor(Mend)` block. Chord follows the TADS division more strictly than TADS does. The action holds grammar and what is true of the player, the thing holds the response, and the compiler will not let either do the other's work.

23.14 What you have so far

The story understands four workings, one old word about sunlight, and Ansel's memory. It knows who may say each working, and how near the thing must be. It does not yet know what any working does. That

knowledge belongs to the lamp, the lens, the bell and the grille, and the next chapter gives it to them.

24 New kinds of thing

The last chapter ended with four verbs and nothing to use them on. The sparking action knows who may spark. It does not know what a flame does to a lamp, and it has been forbidden to find out.

That knowledge goes in a **trait**. You have used traits since chapter 6 without defining one: `openable`, `wearable` and `light-source` are trait words, and each makes a thing answer some standard action. A trait of your own makes a thing answer an action of your own. This chapter defines five of them, and at the end of it The Relighting can be won.

24.1 A trait that answers a verb

Three things in Lanternhall take a flame: the hand lamp, the refectory candles and the Great Lantern. What happens when one of them is sparked is the same in each case, so it is written once.

```
define trait wick
  on the player sparking
    refuse when it is lit: already-burning
    change it to lit
    phrase sparked
  end on
end trait
```

A trait is a block that opens with `define trait` and a name and closes with `end trait`. Inside it is a clause, of exactly the kind you have been writing in `create` blocks since chapter 14. The difference is the word `it`.

A clause in a thing's block names the thing, because the block has one owner. A trait has many, or will have. So inside a trait, `it` means

whichever thing has the trait and is being acted on. This is the one place in Chord where `it` is allowed, as chapter 15 promised.

A thing takes the trait by adding its name to the kind line.

```
create the hand lamp
  a thing, light-source, starts unlit, wick
  in the Lodge

  A tin hand lamp with a wire handle. Its wick is dry and its
  ↪ glass is
  clean, which is Hobb all over.

  phrase detail while the hand lamp is lit:
    It is burning with a small flame, and the flame is blue.
```

Your trait word sits beside the platform's, and reads the same way. The lamp's own `on the player lighting` clause, with its requirement about the page, is gone. The sparking action carries that requirement now.

```
> spark lamp
You say the word. A flame stands up on the wick, blue at first, as
if it had been waiting there with its coat on.

Somewhere above you, old timber takes a weight it had forgotten.

> spark lamp
It is burning already, and the word slides off it.

A slow draught moves through the Hall, one way and then the other.
```

Follow the command through, because this is the centre of the language. The player types `spark lamp`. The sparking action checks its own lines: the lamp is reachable and the page has been learned. It then asks the lamp for an answer. The lamp has the `wick` trait, and the trait has a clause for sparking, so that clause runs with `it` meaning the lamp. Had the player named the grimoire, there would be no such clause, and the action's `otherwise refuse` line would speak.

The action never learned what a lamp is. The lamp never learned what the player must know. Each holds its own half.

The line `change it to lit` deserves a second look. `lit` is not a state you declared. It belongs to `light-source`, and chapter 13 said that such states are read and changed like your own. Here is where that matters. A working that could only set states of your own invention could never light a real lamp, and the Undercroft would stay dark.

24.2 One trait, several things

The candles and the Great Lantern take the same word.

```
create the candles
  aka prickets
  a thing, scenery, plural, light-source, starts unlit, wick
  in the Refectory
  score supper-lit worth 5

  Stubs of tallow in iron prickets down the middle of each table.

  after the player sparking
    move the unbind page here
    award supper-lit
    phrase candles-catch
  end after
```

The candles' `after` clause used to say `lighting`. It now says `sparking`, and it is still in the candles' own block. This is the half of the rule that the last chapter's error message spelled out. A thing may not answer one of your actions with an `on` clause of its own, and it may react to one with an `after` clause. The trait says what sparking does to any wick. The candles say what follows when these particular wicks catch.

```
> spark candles
You say the word. A flame stands up on the wick, blue at first, as
if it had been waiting there with its coat on.

The flame runs down the length of the table, and every cold wick
takes it. In the candlelight there is a leaf of paper under the
middle pricket that the daylight did not show.

The tower bell tolls.

You have risen to the rank of a Second-Year.
```

24.3 A trait may answer standard actions too

There is a hole in this. `light lamp` is a standard command, the lamp is a `light-source`, and the standard lighting action would light it without any word at all. Until now the lamp's own clause stood in the way. The trait can do the same for every wick at once.

```
define trait wick
  on the player sparking
    refuse when it is lit: already-burning
    change it to lit
    phrase sparked
  end on

  on the player lighting
    refuse light-needs-the-word
  end on
end trait
```

```
> light lamp
You have no match, and you did not walk here for three days to use
one. There is a word for this.

The tower bell strikes once, and the sound goes through the stones
under your feet.
```

A trait may hold clauses for any actions, standard or your own, and several of each kind. The next chapter closes this hole in a better way, by teaching the story that `light` means spark.

24.4 A trait with one thing

The Great Lantern needs one thing more than the lamp does. It must not be sparked without its lens. That used to be a `must` line in the Lantern's own `on the player lighting` clause, and a thing's own `on` clause cannot answer sparking.

So the requirement becomes a trait, even though only the Lantern will ever have it.

```
define trait lens-seat
  on the player sparking
    the lens must be in the Great Lantern: lantern-empty
  end on
end trait
```

```
create the Great Lantern
  aka lantern
  a container, scenery, light-source, starts unlit, lens-seat,
  ↪ wick
  in the Lantern Gallery
  holds 1
  score relit worth 15

  A drum of brass and glass the height of a door, with a seat
  ↪ inside it
  for a lens and a wick as thick as your arm. Nothing has burned
  ↪ in it
  for thirty years.

  after the player sparking
    lose lens-shatters when not lantern-ready
    change the story to awake when lantern-ready
    award relit when lantern-ready
    win lantern-lit when lantern-ready
  end after
```

A trait with a single carrier is ordinary in Chord and nothing to be tidied away. It is how a thing answers one of your verbs in a way no other thing does.

The Lantern has two traits that answer sparking, and both of them answer. Their clauses are taken in the order the trait words stand on the kind line, in two passes. First every clause's refusals and requirements are settled. Only when none of them refuses does any clause's body run. So an empty Lantern is refused before any flame is lit, and that is true whichever trait is written first.

What the order decides is what the player reads. If two traits would both refuse, the refusal of the one written first is the one that is spoken. When the command goes through, the bodies run in the same order, and so do their phrases.

```
> spark lantern
There is a wick, and there is a seat for a lens with no lens in
it. A flame here would light the roof and nothing else.
```

24.5 A trait with states and a score

A trait can own states. The lens has been `cracked` or `sound` since chapter 13. That pair is really a fact about anything that can be mended, so it moves into the trait.

```
define trait mendable
  states: cracked, sound
  score made-whole worth 5
  phrases en-US
    mended:
      You say the word and mean it. What was broken is whole, and
      ↪ has
      no memory of having been otherwise.

  on the player mending
    refuse when it is sound: already-whole
    change it to sound
    award made-whole
    phrase mended
  end on
end trait
```

Each thing with the trait gets its own copy of the states. Inside the trait they are read as `it is sound`. Outside, they are read by the thing's name, as before: `the lens is sound`. The `lantern-ready` condition and the lens's `detail` phrases did not have to change.

The lens loses its own `states` line and gains a word.

```
create the lens
  a thing, mendable
  in the crate
```

The trait also holds a `score` line, and a `phrases` block. The block declares a phrase that belongs to the trait, in the list form of `define phrases`, indented inside the trait. It keeps the trait's words beside the trait's clause.

24.6 A thing's own words, again

A second thing in Lanternhall has been broken since chapter 6. Hobb's spectacles have one arm held on with thread.

```
create the spectacles
  aka glasses
  a thing, plural, concealed, wearable, mendable
  in the Lodge

  Wire spectacles. Hobb has been looking for them since the
  ↪ spring.

  phrase detail while the spectacles is cracked:
    One arm has come away, and is held on with a turn of thread.

  phrase detail while the spectacles is sound:
    Both arms are sound.
```

One word on the kind line, and the mend working mends them.

The replies in this section were recorded with the bell already hushed, which the next section does. Mend anything while it is still tolling and the reply ends with the bell.

```
> mend spectacles
You say the word and mean it. What was broken is whole, and has no
memory of having been otherwise.
```

That is the trait's own phrase. It serves for spectacles, and it is too plain for the lens, which is the mend the whole game has been leading up to. Chapter 21 said that a `phrase` line in a thing's block is that thing's own version of a piece of text. Here is the general use it promised.

```
create the lens
  a thing, mendable
  in the crate

  A disc of glass as wide as a cartwheel, ground in rings.

  phrase mended:
    You say the word and mean it. The crack closes from both edges
    ↪ at
    once, like a mouth deciding not to speak.
```

The trait's clause says `phrase mended`. For the spectacles that is the trait's text. For the lens it is the lens's.

```
> mend lens
You say the word and mean it. The crack closes from both edges at
once, like a mouth deciding not to speak.

> examine lens
A disc of glass as wide as a cartwheel, ground in rings. There is
no flaw in it anywhere, and you looked.
```

The score was declared once, in the trait, and it is paid once. Mending the spectacles earns the five points, and mending the lens afterwards earns nothing more. If each mend should score, give each thing a `score` line of its own and award it in an `after the player mending` clause.

24.7 The bell and the grille

Two traits remain, each with one carrier.

```
define trait hushable
  score quieted worth 5

  on the player hushing
    it must be tolling: already-quiet
    change it to silent
    award quieted
    phrase bell-hushed
  end on
end trait

define trait ward
  score unsealed worth 5

  on the player unbinding
    refuse when the tower bell is unrung: ward-asleep
    refuse when the tower bell is tolling: ward-in-the-bell
    change it to lifted
    move it offstage
    award unsealed
    phrase ward-lifts
  end on
end trait
```

The tower bell becomes `a thing, scenery, hushable` and the iron grille `a thing, scenery, ward`. Neither trait declares states. They use `tolling`, `silent` and `lifted`, which the bell and the grille declare themselves, and the compiler checks that every thing with the trait has the states the trait changes.

The `ward` trait is where the puzzle is. The grille cannot be unbound before the bell has rung, because the ward is asleep, and it cannot be unbound while the bell is tolling, because the ward is in the bell. So the bell must be hushed first, and hushing needs the third page.

```
> unbind grille
The word goes into the iron and comes back out on the next stroke
of the bell. Whatever holds the grille is being rung into it.

The bell tolls. You have stopped counting.

> hush bell
You say the word up into the dark of the tower. The bell swings
twice more without a sound, and hangs still.

> unbind grille
You say the word. The grille is not there, in the manner of a
thing that was never there, and the stair goes up.

You have risen to the rank of a Journeyman.
```

The bell has stopped at last. Every turn since chapter 19 has ended with it. The line about rank comes on whichever of these awards takes the score to thirty. Here that was the grille, because the lamp, the candles and the other pages had been seen to first and nothing had been mended yet. In another order it follows the bell, or a mend.

The grille is moved offstage, the Bell Stair's `{grille-across}` marker falls silent because its condition is no longer true, and the way up is no longer blocked. Three lines written in three different chapters agree, because all three read one state.

24.8 Data on a trait

A trait may give each of its things a value of its own, called a **field**. The Relighting has no need of one, so here is a brazier from another story.

```
define trait burner
  data
    fuel: one of tallow, oil
    keeper: optional entity

  on the player stoking
    phrase fuel-tallow when its fuel is tallow
    phrase fuel-oil when its fuel is oil
  end on
end trait
```

The `data` block lists the fields. Each is a name, a colon and a type.

Type	Holds
`one of` and a list	one word from the list
`number`	a whole number
`entity`	a thing, a person or a room
`name`	a word of your choosing

A thing sets its fields on the kind line, after the trait word, with `with` and then `and` between one field and the next.

```
create the cresset
  a thing, burner with fuel oil
  in the Yard

create the brazier
  a thing, burner with fuel oil and keeper the stub
  in the Yard
```

Inside the trait a field is read as `its fuel`, and from outside as `the brazier's keeper`. A field is changed with `set`, as in `set its brightness to 3`.

Set every field on every thing that has the trait, unless the field is marked `optional` and the trait never reads it without checking. A field that was never set has no value to read.

Reach for a field when things with the trait differ by a value. Reach for a state when they differ by a condition that changes during play.

24.9 Is it a wick?

A trait name can be used in a condition, after `is a`.

```
define condition cold-wick: it is a wick and it is not lit
```

`the hand lamp is a wick` is true, and `the kettle is a wick` is false. With an open condition such as `cold-wick`, the `any`, `no` and `each` of chapter 16 can now talk about every thing of a sort you defined.

Because a trait name is used this way, it cannot be a word the language already has. You may not call a trait `container` or `openable`.

24.10 Build and try it

Copy `relighting.story` from the companion folder `chapter-24` over your own file. It is the story with this chapter's changes made, and it holds the phrases, which the chapter did not print in full. To make the changes by hand instead, work from this list and take the rest from that file.

Add the five traits above the actions. Give `wick` to the lamp, the candles and the Lantern, with `lens-seat` before it on the Lantern. Give `mendable` to the lens and the spectacles, `hushable` to the bell and `ward` to the grille. Remove the three `on the player lighting` clauses from the lamp, the candles and the Lantern, and change the candles' and the Lantern's `after` clauses to `sparking`. Remove the lens's `states` line. Lengthen the rank ladder to its final form: Journeyman at 30, Lamplighter at 50. There are nine new phrases.

```
sharpee compose relighting.story --check
sharpee play .
```

The game can be played to its end for the first time. Read the first page. Spark the lamp and the candles. Find the three other pages and read them. Hush the bell, unbind the grille, mend the lens, carry it to the roof and seat it.

```
> spark lantern
You say the word. A flame stands up on the wick, blue at first, as
if it had been waiting there with its coat on.

The lens gathers the flame and sends it out over the valley in one
white bar. Down in the Hall, every blank page begins to fill.

You have risen to the rank of a Lamplighter.

[The story has ended. RESTART, RESTORE or UNDO]
```

Then play it again and leave the lens cracked.

UNDER THE HOOD: CAPABILITY · STORY IR `TRAITS`

Find `wick` under `traits` in the compiled IR and look at its sparking clause. One of its fields reads `"routing": "capability"`. That is the platform's name for what this chapter has been doing. A trait declares that it is capable of answering an action, and the action asks every trait of the thing that answers it, in the order they are written. The standard traits answer the standard actions through the same door. Your `wick` and the platform's `openable` are the same kind of object, which is why they sit side by side on a kind line.

FROM ANOTHER SYSTEM: KINDS AND CLASSES

In Inform 10 you would say "A wick is a kind of thing" and write rules for sparking a wick. In TADS 3 you would define a `Wick` class and give it a `dobjFor(Spark)` block, and an object could list several classes. Chord's trait is nearer to the TADS class used as a mix-in. A thing has exactly one kind, such as `a thing` or `a container`, and any number of traits. Traits do not inherit from one another. What several traits share, they share by being composed on the same thing.

24.11 What you have so far

The four workings work. A thing answers a working by having a trait, and each trait says in one place what the working does. The story can be won and lost, and the bell can be stopped.

A player who types `light lamp` is still told off for it, though, and one who types `fix lens` or `sit on chair` is not understood at all. The last chapter of Part IV makes the story's grammar as forgiving as its prose.

25 Bending the standard grammar

A player does not know which words you chose. They will type `fix lens` where you wrote `mend`, `sit on chair` where the platform knows `enter`, and `light lamp` where the whole point of your story is that lamps are sparked. A story that answers “I don’t understand that” to a reasonable sentence has lost a little of the player’s trust each time.

This chapter is about the grammar of actions that already exist: adding words to them, taking words away, and moving a word from one action to another. It is the shortest chapter in Part IV, and it has a large effect on how a story feels to play.

25.1 Adding words to an action

`extend action` adds patterns to an action. The action may be one of your own.

```
extend action mending
  grammar
    fix or repair the target

extend action hushing
  grammar
    quiet or silence the target
```

The block names the action and holds a `grammar` block, written as in `define action`. Now `fix`, `repair` and `mend` are one verb, with one set of requirements and one trait answering it.

```
> fix spectacles
You say the word and mean it. What was broken is whole, and has no
memory of having been otherwise.
```

You could have written all three words in the action's own `grammar` block, and for a new action that is often tidier. `extend action` earns its place when the action is not yours to edit.

25.2 Extending a standard action

Hobb's chair has been `enterable` since chapter 7, and chapter 7 had to admit that the command for sitting in it was `enter chair`. In the story as chapter 24 left it, the plainer command gets this reply:

```
> sit on chair
I don't understand that.
```

The standard action is called `entering`, and it can be extended like any other.

```
extend action entering
  grammar
    sit on or in the portal
    sit down on or in the portal
```

With that in the story, which is how this chapter's companion file has it:

```
> sit on chair
You get into the chair.

> exit
You get out of the chair.
```

The slot is called `the portal` here because that is what the standard entering action calls it. When you extend a standard action, use its own slot names, so that the rest of the action knows which thing your pattern means. Appendix C gives the patterns of every standard action, and the slot names are there to be read off: `the item` for taking, `the target` for lighting, `the portal` for entering.

The reply still says “get into”, which is the standard message `entering-entered`. You know from chapter 22 how to change that.

An `extend action` block adds grammar and nothing else. It cannot add a requirement, a refusal or a statement to the action. Behaviour belongs to things and traits, by the rule of the last two chapters, and the compiler refuses a behaviour line in an `extend action` block.

25.3 Taking words away

`remove from action` takes a pattern out of a standard action.

```
remove from action lighting
  light the target
```

The block names the action, and each line beneath is a pattern to remove, written exactly as the standard grammar has it. The pattern must be one the action really has. Removing a pattern that is not there is an error and not a silent success, and the message lists the patterns the action does have.

```
relighting.story:192:1 error [analysis.unmatched-removal-pattern]
`remove from action lighting` — no standard rule matches
`ignite :target`. The action's standard patterns are:
`light :target`, `light :target with|using :tool`.
```

After this block, `light lamp` is not a command at all. The standard lighting action still exists, with its other pattern, `light the target with the tool`. It has lost only the words you named.

By itself that would make the story worse. It is one half of a move.

25.4 Moving a word

The last chapter left a hole and a patch. `light lamp` reached the standard lighting action, and the `wick` trait refused it with a sentence

about matches. That is a fair answer, and a better one is to let the player be right. In this story, to light a thing is to spark it.

```
remove from action lighting
  light the target

extend action sparking
  grammar
    light the target
```

The first block frees the pattern. The second gives it to sparking. Write both. An `extend action` block is allowed to add a pattern that a standard action also has, and the story would then have two actions claiming one command. Removing the pattern first leaves no doubt about which you mean.

```
> light lamp
You say the word. A flame stands up on the wick, blue at first, as
if it had been waiting there with its coat on.

Somewhere above you, old timber takes a weight it had forgotten.
```

And before the page has been read, `light lamp` is refused by the sparking action, in the words the story uses for a working that has not been learned.

```
> light lamp
You reach for the word and find only the place where it was.
```

The `wick` trait keeps its `on the player lighting` clause. It now guards the one pattern the lighting action has left.

```
> light lamp with taper
You have no match, and you did not walk here for three days to use
one. There is a word for this.

A slow draught moves through the Hall, one way and then the other.
```

This is the right order in which to think about any standard verb that your story changes the meaning of. First decide what the player will type. Then make sure every reasonable way of typing it arrives at the same action.

25.5 What to add

Play your story as someone who has never seen the source, or better, watch someone else play it. Each time a sensible command is not understood, write it down. Most of the entries on that list are one line in an `extend action` block.

A few habits help.

Add the plain word beside the coloured one. Your verb is `spark`, and the player's is `light`. Your verb is `unbind`, and theirs may be `open`. The Relighting does not give `open` to unbinding, because the grille has no door in it and "open grille" deserves the standard refusal, but it is the kind of question to ask of every working.

Add the prepositions people actually use. `sit on`, `sit in` and `sit down on` are one pattern each, or one pattern with `or` in it.

Do not add a synonym that means something else elsewhere in the story. If `ring` is a noun in your game, think before making it a verb.

And do not take away a standard pattern merely because your story has no use for it. A command that is understood and politely refused is better than one that is not understood.

25.6 A file for the grammar

The synonym blocks are not part of the story's design. They are a list that grows with testing, and they can live in a file of their own.

```
import "synonyms"
```

This is the `import` line of chapter 22, and the file it names is `synonyms.chord`, beside the story.

```
## Plain words for the workings, and a way to sit down. This file
↪ is a
## fragment: The Relighting imports it, and it is not a story by
↪ itself.

extend action mending
  grammar
    fix or repair the target

extend action hushing
  grammar
    quiet or silence the target

extend action entering
  grammar
    sit on or in the portal
    sit down on or in the portal
```

The blocks are read as though they stood in the story at the line where the `import` is. The Relighting keeps the `light` move in the main file, because that one is a decision about what the story means, and puts the rest in the fragment.

The standard grammar is itself a file of this kind, written in the same `define action` blocks you learned in chapter 23. Appendix C prints it in full. Reading it is the quickest way to learn what your players can already say.

25.7 Build and try it

Copy `relighting.story` and `synonyms.chord` from the companion folder `chapter-25` into your project, the first over your own file and the second beside it. They are the story with this chapter's changes made. To

make the changes by hand instead, work from this list: add the `remove from action` block, the `extend action sparking` block and the `import` line near the workings, and create `synonyms.chord` beside the story with the three blocks above.

```
sharpee compose relighting.story --check
sharpee play .
```

Play the winning walk once more with the plain words: `light lamp`, `quiet bell`, `fix spectacles`, `repair lens`. The grille has no plain word, so `unbind grille` is typed as before. Leave it until after `fix spectacles`.

```
> quiet bell
You say the word up into the dark of the tower. The bell swings
twice more without a sound, and hangs still.
```

One of the replies on this walk ends with the line about the rank of a Journeyman. It is whichever takes the score to thirty, and in the order given here that is `fix spectacles`.

UNDER THE HOOD: TWO LISTS · STORY IR `GRAMMARREMOVALS`, `GRAMMAREXTENSIONS`

The compiled IR records these blocks in two lists. `grammarRemovals` has one entry, for lighting, holding the pattern that was taken away. `grammarExtensions` has four, for sparking, mending, hushing and entering, and it does not say which file they came from. When the story loads, the platform starts from the standard grammar, removes the first list, and adds the second and your own actions. The parser that reads the player's command never sees your source. It sees one grammar, which is the standard one as your story has bent it.

FROM ANOTHER SYSTEM: UNDERSTAND LINES

In Inform 10 the same three jobs are done by "Understand" sentences: "Understand 'fix [something]' as mending" adds a pattern, and "Understand the command 'light' as something new" clears a verb so that you can assign it again. In TADS 3 you add a `VerbRule` for the new wording. Chord's two blocks correspond closely to the two Inform forms. The difference is that Chord removes one pattern at a time, where Inform's "as something new" clears every use of the word.

25.8 What you have so far

That is Part IV, and with it the game has its centre. The Relighting has four workings that can be typed three or four ways each, things that answer them through traits, prose that varies and belongs to its speakers, and a standard library that has been taught this story's manners. A player can win it, lose it two ways, or run out of day.

What Lanternhall does not have is anybody in it. Hobb stands in his doorway and says a word about sunlight when you come in, and he cannot be asked a single question. Part V is about people.

PART V. PEOPLE

26 People who act

Hobb has been in the Lodge since chapter 6. He is `a person`, he is `proper`, and he has a description and a sentence of his own in the room. In every way that matters to the player he is furniture. He never moves, he never does anything, and the only verb he answers is `give`.

This chapter makes people do things. It covers four tools, in rising order of effort: the `pronouns` line, which lets your prose refer to a person correctly; the stock behaviours, which move a person about with one word; the acting statement, which has a person perform one action when you say so; and the `goal` block, which gives a person a plan to carry out on their own turns.

26.1 Pronouns

A person takes one line that a thing does not.

```
create Hobb
  a person, proper
  pronouns he
  in the Lodge
```

The `pronouns` line names a pronoun set. Four are standard: `he`, `she`, `it` and `they`. The line is allowed only in a person's block.

What the line buys you is a marker. Chapter 20 showed that a phrase can take a parameter and print it with `{the who}`. Once a person has been named in a phrase, `{pronoun:subject}` prints the right pronoun for whoever that was.

```
define phrases en-US
  own-business:
    {the who} does not stop for you. {capitalize pronoun:subject}
    ↪ has
    {pronoun:possessive} own day to get through.
```

One phrase now serves anybody. The person is passed in where the phrase is used.

```
after the player examining
  phrase own-business with who = Tamsin Vey
end after
```

```
> examine tamsin
A broad woman in a leather apron, with a pry bar through her belt
and brass filings in the creases of her hands. She has the look of
someone adding you up.

Tamsin Vey does not stop for you. She has her own day to get
through.
```

A pronoun marker refers to the person most recently mentioned in the same phrase, which is why `{the who}` comes first. There are five cases, and each has a marker.

Marker	For `he`	For `she`	For `it`	For `they`
`{pronoun:subject}`	he	she	it	they
`{pronoun:object}`	him	her	it	them
`{pronoun:possessive}`	his	her	its	their
`{pronoun:possessive-pronoun}`	his	hers	its	theirs
`{pronoun:reflexive}`	himself	herself	itself	themselves

Put `capitalize` in front of a marker that begins a sentence. A person with no `pronouns` line is spoken of as `it`, or as `they` if the person is `plural`.

26.1.1 A set of your own

If the four standard sets do not suit a character, define a fifth.

```
define pronouns ey
  subject ey
  object em
  possessive eir
  possessive-pronoun eirs
  reflexive emself
end pronouns

create Sam
  a person, proper
  pronouns ey
  in the Yard
```

The block names the set and gives all five cases, one to a line, in any order. Leaving one out is an error. The name may not be one of the four standard names.

```
> examine sam
Sam.

Sam looks back. Ey folds eir arms and keeps the moment to emself.
```

The Relighting does not need a set of its own. Hobb is `he`, Tamsin Vey is `she`, and the third person in this chapter is `it`.

26.2 The stock behaviours

The quickest way to make a person move is one word on the kind line. The platform ships five behaviours, and each is a word you compose exactly as you compose `wearable` or `openable`.

The Relighting has a creature for the simplest of them: a scrap of returning magic that takes a liking to Ansel.

```
create the wisp
  aka light
  a person, follower
  pronouns it

  A scrap of light the size of a thumbnail, the same blue as the
  ↪ lamp
  flame. It keeps a little behind your shoulder, where a dog
  ↪ would.
```

A `follower` goes where the player goes. The wisp has no placement line, so it begins the game nowhere, and the hand lamp brings it on the first time the lamp is sparked.

```
after the player sparking, once
  move the wisp here
  phrase wisp-arrives
end after
```

```
> light lamp
You say the word. A flame stands up on the wick, blue at first, as
if it had been waiting there with its coat on.

Something small and blue comes loose from the flame and hangs in
the air beside it, undecided. Then it decides.

Somewhere above you, old timber takes a weight it had forgotten.

> east
Cloister
A covered walk around a garth gone to nettles. The refectory is
east, the library door west, and the foot of the bell stair north.
Worn steps go down into the dark under the hall.

A slow draught moves through the Hall, one way and then the other.

The wisp follows you.
```

The line “The wisp follows you.” is the platform’s. From here to the end of the game the wisp is in the room with Ansel, and a `look` lists it. On

the turn Ansel walks into a room, the room's list is printed before the wisp arrives, so the wisp is not in that one.

A person needs the word `person`. The wisp is not much of a person, but it moves by itself and will later hear things, and those are what the kind is for.

26.2.1 The five words

Word	What the person does
`follower`	Goes where the player goes, arriving on the same turn.
`wanderer`	Moves to a neighbouring room at random.
`patrol`	Walks a fixed route of rooms.
`guard`	Stays where it is, and says so when the player walks in.
`passive`	Stays where it is and does nothing.

A behaviour word takes settings, written with `with` and joined by `and`, as the settings of chapter 9 were.

```
create the cat
  a person, wanderer with move-chance 50 and allowed-rooms [the
  ↪ Yard, the Shed]
  in the Yard

  A grey cat.

create the watchman
  a person, patrol with route [the Yard, the Shed] and loop true
  ↪ and wait-turns 1 and announces-movement true
  pronouns he
  in the Yard

  A watchman.
```

A list of rooms is written in square brackets, with commas. A setting that is true or false takes the word `true` or `false`.

Setting	On	Meaning
`move-chance`	`wanderer`	The chance of moving each turn, as a number from 0 to 100.
`route`	`patrol`	The rooms to walk, in order.
`loop`	`patrol`	Whether to begin the route again at its end.
`wait-turns`	`patrol`	How many turns to stand in each room.
`allowed-rooms`	all five	The only rooms the person may enter.
`forbidden-rooms`	all five	Rooms the person may never enter.
`can-move`	all five	Set to `false`, it holds the person still.
`announces-movement`	all five	Whether the player is told of comings and goings.

The last one matters more than it looks. Without it a person who moves is simply in the room or not in it, and the player learns of the move from the room's list. With it, the platform reports each move the player can see.

```
> look
Yard
A yard.

You can see a watchman here.

The watchman leaves to the north.
```

Two of the behaviours speak when the player arrives. A wanderer "notices you", and a guard "blocks your way!". The guard's line is only a line: the

player has already walked in. A guard that really stops someone is a blocked exit, which is chapter 12.

26.3 Having a person do something

The stock behaviours move people. To have a person *do* something, write a sentence with the person as its subject.

```
on every turn while the oak press is open and the library key is
↪ not in the oak press
  Hobb closes the oak press
end on
```

`Hobb closes the oak press` is the **acting statement**. It begins with a person's name, and the rest is a command exactly as a player would type it, with the verb in the third person. When the statement runs, Hobb performs the closing action. It is the same action the player's `close press` reaches, with the same checks, and the platform reports it.

```
> take key
You take the library key from the oak press.

Hobb closes the oak press.

> open press
You open the oak press, which is empty.

Hobb closes the oak press.
```

Hobb is a porter. He has spent thirty years shutting things that other people left open, and now the story says so in one line.

Three rules govern the statement.

The words after the name must be a command the story understands. Any standard action will do, and so will an action of your own from

chapter 23: `Hobb opens the oak press`, `Nell takes the coin from the box`. A verb that no grammar has is an error when you build.

The person performs the action and does not merely have its result. This is the difference between `move` and acting, and it is worth a moment. `move the library key to Hobb` puts the key in his hands by decree, from anywhere, with no report. An acting statement has the person do the thing as a player would, under the action's own checks, and the platform tells the player it was done.

The statement is not for the player, whose actions are the player's to type.

```
probe.story:23:5 error [analysis.act-player-actor] `the player
takes the bucket` — the player cannot be made to act; an acting
statement names a character. Write the character's own name, or
use `move` for an authorial change.
```

It also cannot stand in an `on the player …` clause, which runs before an action and exists to permit or refuse it. Write it in an `after` clause, in `on every turn`, in a `when` clause, or in the conversation rows of the next two chapters. A `when` suffix is allowed, as on most statements.

26.4 Goals

An acting statement is one action at a moment you choose. A person who goes somewhere, does three things in order and then stops needs a plan, and a plan is a `goal` block.

The Relighting's second person is a salvager.

```
create Tamsin Vey
  aka salvager, woman
  a person, proper, passive with announces-movement true
  pronouns she
  in the Refectory

  A broad woman in a leather apron, with a pry bar through her
  ↪ belt and
  brass filings in the creases of her hands. She has the look of
  ↪ someone
  adding you up.

  goal rounds, low
    wait for Ansel is in the Refectory
    act tamsin-sorts
    move to the Cloister
    act tamsin-measures
    move to the Refectory
  end goal
```

A goal has a name and a priority, and its body is a list of steps. The person takes one step on each turn, after the player's own command. This goal waits until Ansel walks in, and then Tamsin goes about her business with an audience.

```
> east
Refectory
Three long tables, each with its row of candles, and a hearth you
could stable a horse in. The benches are stacked against the wall,
where someone left them for the holidays. The cloister is back to
the west.

You can see Tamsin Vey here.

> examine tamsin
A broad woman in a leather apron, with a pry bar through her belt
and brass filings in the creases of her hands. She has the look of
someone adding you up.

Tamsin Vey does not stop for you. She has her own day to get
through.

Tamsin Vey turns over a heap of door furniture on the end of a
table, and puts two hinges in her apron.

> wait
You wait. The Hall has had thirty years of practice and is better
at it than you are.

Tamsin Vey leaves to the west.
```

The line “Tamsin Vey leaves to the west.” is there because of `announces-movement`. That is the reason she is `passive`: the word gives her no movement of its own, and it is a place to hang the setting.

26.4.1 The steps

Step	What the person does
`move to the <room>`	Walks to the room, one room a turn.
`seek the <thing>`	Walks toward a thing or a person, one room a turn.
`wait for <condition>`	Stays on this step until the condition holds.

Step	What the person does
`act <phrase>`	Prints the phrase, if the player is there to see.
`say <phrase>`	Prints the phrase as something said. `say <phrase> to <person>` names the hearer.
`acquire the <thing>`	Waits until the thing is in the room, then takes it.
`give the <thing> to <person>`	Hands it over, waiting if the other person is not there.
`drop the <thing>`	Puts it down.
a command	Performs that action, as the acting statement does.

The last row is the general one. Any command the story understands may be a step, written without the person's name, since the goal's owner is the one acting: `open the box`, `go east`, `push the iron grille`. If the action is refused, the person tries again on the next turn.

A step has no `when` suffix. A step that must wait for something is preceded by a `wait for` step.

Here is a small complete plan, from a story of two rooms.

```
create Nell
  a person, proper
  pronouns she
  in the Yard

  Nell.

  goal fetch, high
    active when Pat is in the Shed
    move to the Shed
    open the box
    take the coin from the box
    act nell-bites
    wait for Pat is in the Yard
    seek Pat
    say nell-done to Pat
  end goal
```

```
> look
Shed
A shed.

You can see a box and Nell here.

Nell opens the box.

In the box you see a coin.

> look
Shed
A shed.

You can see a box and Nell here.

In the box you see a coin.

Nell takes the coin from the box.

> look
Shed
A shed.

You can see a box and Nell here.

Nell bites the coin.
```

`acquire` takes a thing that is lying loose in the room. For a thing inside something, write the command, as Nell's plan does.

A person's steps happen whether or not the player is watching. Phrases printed by `act` and `say` are seen only by a player in the same room, and a goal that runs to its end in an empty room has still run. A goal is finished when its last step is done, and it does not begin again.

26.4.2 When a goal is active

A goal may carry one `active when` line. The goal is considered only while its condition holds.

```
goal the-grille, high
  active when the tower bell is tolling
  move to the Bell Stair
  say tamsin-at-the-grille
  push the iron grille
  act tamsin-tries-the-bar
end goal
```

Tamsin now has two goals. The priority, which is one of `critical`, `high`, `medium` and `low`, decides between goals that are active at once. While the Hall sleeps only `rounds` is active. When the bell begins to toll, `the-grille` becomes active and outranks it, and Tamsin leaves whatever she was doing and walks to the foot of the tower. She takes a step a turn, so what you see depends on when you follow. This is the view two turns after she comes through the Cloister.

```
> north
Bell Stair
The foot of the tower. A stair winds up around the well where the
bell rope hangs. The cloister is south. An iron grille stands
across its third step.

You can see Tamsin Vey here.

The tower bell tolls.

The wisp follows you.

Tamsin Vey runs a thumb down one bar of the grille and looks at
the thumb. "That," she says, "was not here on Tuesday."

> wait
You wait.

The bell tolls again, and the rooks go up off the roof.

Tamsin Vey pushes the iron grille.

> wait
You wait. The Hall has had thirty years of practice and is better
at it than you are.

The bell tolls. You have stopped counting.

Tamsin Vey sets her pry bar to the foot of the grille and leans.
The bar bends. The grille does not notice.
```

The middle line is the platform's report of a real push. The grille's own clause from chapter 14 says `on the player taking, pushing, pulling`, and it concerns the player only, so Tamsin's push goes through to the standard action and its standard result.

That is the whole of a second character's morning in fourteen lines, and none of it is a clause on a thing. A goal belongs to the person who has it.

26.5 Build and try it

Copy `relighting.story` from the companion folder `chapter-26` over your own file. It is the story with this chapter's changes made, and it holds the phrases, which the chapter did not print in full. This is the first chapter of Part V, and the copy's `story-version` is `0.5.0`. To make the changes by hand instead, work from this list and take the rest from that file: add `pronouns he` to Hobb, add the `on every turn` clause to the oak press and the `after the player sparking, once` clause to the hand lamp, create Tamsin Vey and the wisp near Hobb, and add their six phrases to the phrases block.

```
sharpee compose relighting.story --check
sharpee play .
```

Walk to the Refectory and wait a turn or two, then follow Tamsin into the Cloister. Read the spark page, light the lamp, and wait in the Cloister for the bell. Tamsin comes through on her way to the tower. Wait two turns more and then go north, and the Bell Stair reads as it does above. Go sooner or later and the same lines come on other turns.

```
> wait
You wait.

The tower bell tolls.

Tamsin Vey enters from the east.
```

UNDER THE HOOD: BEHAVIOURS AND PLANS · STORY IR `TRAITS`, `CHARACTER.GOALS`

In the compiled IR a behaviour word is one more entry in a person's `traits` list, beside `proper`, with its settings in a `config` list: Tamsin has `passive` there with `announces-movement`. The platform turns each into an NPC behaviour that is consulted once a turn. A person's goals are recorded under `character.goals`, each with its `priority`, its `activeWhen` condition and its `steps`. A command step such as `push the iron grille` is stored with the action it resolved to (`pushing`) and the pattern it matched (`push :target`), which is why a misspelt command in a goal is a build error and not a silent failure at turn forty. When an NPC acts, it runs the same standard action a player's command runs.

FROM ANOTHER SYSTEM: ACTORS ACTING

In Inform 10 an action can be tried by another person, with a phrase such as "try Hobb closing the oak press", and the action is then subject to the usual action machinery. A person's routine can be written as "Every turn" rules. In TADS 3 a non-player character's goals are `AgendaItem` objects, each with an `isReady` condition and an `invokeItem` method that is called when the item is carried out. A Chord goal is nearest to a TADS agenda item, with the difference that its body is a list of steps taken one a turn and not a method.

26.6 What you have so far

Lanternhall has three people in it besides Ansel. Hobb shuts his press behind you. Tamsin Vey sorts her brass, measures a bracket in the Cloister, and goes to the grille when the bell starts. A wisp follows the lamp.

None of them can be spoken to.

```
> talk to hobb
Hobb doesn't respond.
```

The next chapter gives them something to say.

27 Asking and telling

The oldest way to talk to a character in interactive fiction is to ask about a subject: `ask hobb about the bell`. It has lasted because it is honest about what the game can do. The player names a subject, the character has something to say about it or has not, and nobody has to pretend the program understands English.

Chord's form of it is a table. Each person has one, and each row of the table is a subject and what follows from raising it. This chapter writes the tables for Hobb and for Tamsin Vey, gives a book a table of its own, and then adds the two things that make a table sound like a person: a greeting and a manner.

27.1 A table of topics

```
define topics for Hobb
  about "dimming":
    phrase hobb-on-dimming
  about "lantern", "great lantern", "light":
    phrase hobb-on-lantern
  about "tamsin", "tamsin vey", "vey", "salvager":
    phrase hobb-on-tamsin
  about "gnusto":
    phrase hobb-on-gnusto
end topics
```

The block is a top-level declaration, like `define action`. It names the person it belongs to, and a person may have only one. Each row begins `about`, gives its subject, and ends with a colon. The body beneath is a list of statements, and a row with one statement may carry it on the same line.

A quoted subject is matched against what the player typed after `about`. The match is of the whole subject and is never a guess: "lantern" answers

`ask hobb about lantern` and does not answer `ask hobb about lanterns`. Give a row every spelling a player might reasonably use, separated by commas.

```
> ask hobb about the dimming
He looks at the blue flame on the sill before he answers.

"Went out like a pipe," says Hobb. "No bang. The Master stood in
the court with his hand up for an hour and nothing came to it."

> ask hobb about tamsin
He takes his time, of which he has had a great deal.

"Vey? Takes the brass and leaves the hinges. I let her. Brass
doesn't keep the rain out." He considers. "Her word's good. It's
all of her that is."
```

The sentence above each answer is one of three that Hobb's manner takes in turn, which a later section of this chapter explains. Which of the three you see depends on how many answers he has given in your game, so in this chapter and the three after it, that sentence in your play will often be a different one of the three from the one the page shows.

Each answer has a sentence in front of it that is not in the phrase. That is Hobb's manner, and this chapter comes to it below.

The player's article is dropped before the match, so "dimming" answers both `about dimming` and `about the dimming`. For the same reason you write a subject once, without its article. Writing it both ways is an error.

```
probe.story:23:9 error [analysis.duplicate-topic] "rain" is
already declared in this table (line 23) — aliases included, a
topic answers in one place.
```

`tell hobb about gnusto` reaches the same table as `ask hobb about gnusto`. A subject has one row, and the row answers either verb.

27.1.1 A thing as a subject

A row may name a thing in the story in place of a quoted word.

```
about the kettle:
  phrase hobb-on-kettle
```

```
> ask hobb about the kettle
He turns the kettle a quarter round on its hook, to no purpose.

"Mine," says Hobb.
```

A row of this kind answers when the player's words name that thing, by any of its names, and the thing is there to be named. The kettle is in the Lodge with Hobb, so the row works. It is the right form for something the player can point at.

It is the wrong form for a subject that is somewhere else. Tamsin Vey is in the Refectory, and a row written `about Tamsin Vey` in Hobb's table would not answer a player standing in the Lodge. That is why Hobb's row about her is quoted, with her names spelled out. People are asked about in their absence more often than not, so quote them.

A row cannot be both. If a table has a row for the thing, a quoted row spelling the thing's name is an error.

```
probe.story:31:9 error [analysis.topic-entity-collision] "bucket"
collides with `bucket` — that entity is already an entity-tier row
of this table, and the quoted spelling would shadow its quiet
entity resolution. Remove one.
```

27.1.2 What a row can do

A row's body is statements, and they are the statements of chapter 15. A row can refuse, change a state, award a score, or have its owner act.

```
about "bell", "tolling", "noise":
  refuse when the tower bell is unrung: hobb-what-bell
  phrase hobb-on-bell
  Hobb gives the hush page to Ansel when Hobb has the hush page
```

This row is how the third working now reaches the player. Until this chapter the hush page lay on a shelf in the stacks. It has moved into Hobb's keeping.

```
create Hobb
  a person, proper
  pronouns he
  in the Lodge
  carries the hush page
  mood calm
```

Ask him before the bell has rung, and the row refuses.

```
> ask hobb about the bell
"Bell hasn't rung in thirty years," says Hobb. "Ask me again when
it does."
```

Ask him when the Hall is ringing with it, and he answers, and then acts.

```
> ask hobb about the bell
He takes one hand off one ear to hear you, and puts it back.

"It'll go on till it's told," Hobb shouts. "There's a word for the
telling. I kept it by, in a newspaper, in case."

Hobb gives the hush page to Ansel.

The bell tolls again, and the rooks go up off the roof.
```

The last line of the row is the acting statement of chapter 26, with a `when` suffix so that he hands the page over once. Ask a second time and he says his piece again and has nothing to give.

27.1.3 A subject nobody has

A subject with no row gets the platform's stock answer.

```
> ask hobb about the weather
Hobb says, "I don't know anything about that."
```

That line is a standard message, and its alias is `asking-unknown-topic`. Chapter 22 showed how to replace one. The Relighting adds it to the `override messages` block it already has.

```
override messages en-US
  asking-unknown-topic:
    {capitalize the target} gives that some thought, and gives it
    ↪ back.
```

```
> ask hobb about the weather
Hobb gives that some thought, and gives it back.
```

The override serves every person in the story. A table has no `otherwise` row of its own.

`ask hobb for the kettle` is a different action from asking about a subject. It is a request for a thing, and with nothing written for it the platform answers "Hobb doesn't seem inclined to hand over the kettle."

27.2 Looking things up

A topic table may also belong to a thing that is `readable`. The player does not ask a book. The player consults it.

```
define topics for the grimoire
  about "gnusto":
    phrase grimoire-on-gnusto
  about "spark", "sparking", "flame":
    phrase grimoire-on-spark
end topics
```

```
> consult grimoire about gnusto
You turn to the margin again. The note is all there is: came over
the water as a working, meaning lost. Whoever wrote it underlined
"lost" twice.

> look up cheese in grimoire
You find nothing about that in the grimoire.
```

The standard grammar has three ways of saying it: `consult the grimoire about gnusto`, `look up gnusto in the grimoire`, and `read about gnusto in the grimoire`. The table is written exactly as a person's is. A directory, a ledger, a dictionary and an index are all this one construct.

27.3 Greetings

A question that has a row is answered at once. Something has to happen when the player opens a conversation with no such question: `talk to hobb`, or a first question he has no row for. That is a greeting.

```
define greetings for Hobb
  first time:
    phrase hobb-greets
  on return:
    phrase hobb-greets-again
end greetings
```

```
> talk to hobb
He turns the kettle a quarter round on its hook, to no purpose.

"It's you," says Hobb. He does not say which you.
```

The `first time` row is used once. After the player has said `goodbye`, or has walked away and come back, the next opening uses the `on return` row.

```
> goodbye
You say goodbye to Hobb.

> talk to hobb
He looks at the blue flame on the sill before he answers.

"Still here," says Hobb, of one of you.
```

`on return` may be narrowed by how long the player was gone: `on return, again so soon`, `on return, after a while`, and `on return, after days`. A plain `on return` covers whichever of those you have not written.

A person with no greetings block has nothing to say to `talk to`, and the platform answers “Tamsin Vey doesn’t respond.” Tamsin has a table and no greeting, and that suits her for now.

27.3.1 A person who is being talked to stays

While a conversation is open, the person in it does not go on with a goal. Tamsin Vey, asked about her brass in the Refectory, answers, and does not turn to her heap of door furniture as she did in the last chapter. She takes her rounds up again when the player has left her.

27.3.2 The line a greeting needs

Greetings, and the manner below, belong to a fuller model of a person than a topic table needs. A person who has either block must also have at

least one line of that model in their own block. Chapter 29 is about those lines. Until then, one is enough, and the plainest is a mood.

```
mood calm
```

Leave it out and the story compiles, and then fails when it is loaded.

```
Error loading story: `Nell` declares conversation blocks but no
character model — scenes need a modeled owner. (line 16)
```

A topic table alone does not need the line. Tamsin already has the model, because a goal is part of it.

27.4 Manner

A character who answers every question in one flat paragraph is a reference book. People do something before they speak.

```
define manner for Hobb
  when the tower bell is not tolling:
    beat "He turns the kettle a quarter round on its hook, to no
    ↪ purpose."
    beat "He looks at the blue flame on the sill before he
    ↪ answers."
    beat "He takes his time, of which he has had a great deal."
  when the tower bell is tolling:
    beat "He takes one hand off one ear to hear you, and puts it
    ↪ back."
end manner
```

A manner is a set of rows, each a condition and one or more beats. A beat is a quoted sentence. When the person greets the player or answers from their table, the platform takes the next beat from a row whose condition holds and prints it as a paragraph of its own, ahead of the line.

The beats of a row are used in turn, so Hobb's three come round one after another and no answer has the beat of the one before it. You have

already seen all three in the transcripts above. A row with one beat prints that beat every time, which is right for a man with his hands over his ears.

A beat is not printed before a refusal or before the stock answer for an unknown subject. Those are not the person speaking in their own voice.

The value of the block is that it is separate from the answers. Hobb has five rows in his table and may have fifteen by the end of the book. None of their phrases needs to mention the kettle.

27.5 Build and try it

Copy `relighting.story` from the companion folder `chapter-27` over your own file. It is the story with this chapter's changes made, and it holds the topic tables, the greetings and the phrases that the chapter did not print in full. To make the changes by hand instead, work from this list and take the rest from that file. Add `carries the hush page` and `mood calm` to Hobb, and take the hush page's `in the Stack 3` line away. Add the greetings, the manner and the three topic tables near the people, the one line to `override messages`, and the new phrases to the phrases block. The story's day is also longer from this chapter on, since conversation takes turns: the sequence's first step is `at turn 70` and its second `40 turns later`.

```
sharpee compose relighting.story --check
sharpee play .
```

Ask Hobb about everything in his table, and about something that is not. Then play on until the bell is tolling, come back to the Lodge, and ask him about the bell.

```
> read hush page
The third working is hardly a word at all. It is the breath you
take before one.

The bell tolls. You have stopped counting.
```

UNDER THE HOOD: WHERE THE TABLE GOES · STORY IR

`TOPICS`, `GREETINGS`, `MANNER`, `CHARACTER`

In the compiled IR a topic table is the `topics` list of the entity that owns it, whether that is a person or a book. Each row has a `filter` and a `body`. A quoted row's filter is of kind `text`, with a `primary` spelling and its `aliases`; the body is the same list of statements a clause has. A person's `greetings` and `manner` are two more keys on the entity, and the `mood calm` line appears under `character`, the key that also holds the goals of chapter 26. The standard actions `asking`, `telling` and `consulting` each take the player's words after `about` as a topic, and all three look the topic up in the owner's one table.

FROM ANOTHER SYSTEM: ASKING ABOUT

Inform 10 has the actions of asking someone about something and telling someone about something, and an author answers them with rules such as "Instead of asking Hobb about 'bell'", or with a table that has a topic column. TADS 3 gives each character a set of topic objects: `AskTopic`, `TellTopic` and `AskTellTopic` for subjects, `DefaultAskTellTopic` for a subject that has no response, `HelloTopic` and `ByeTopic` for greetings and partings, and `ConsultTopic` for a book. A Chord topic table is close to the TADS arrangement, written as rows of one block where TADS has an object for each.

27.6 What you have so far

Hobb can be asked about the Dimming, the Lantern, Tamsin Vey, his kettle, the word gnusto and the bell, and when the bell is tolling he gives up the page that quiets it. Tamsin answers three questions. The grimoire can be consulted. The game can still be won, and the walk to the win now goes by way of the Lodge.

What a table cannot do is hold a thread. Every question stands alone, and Hobb has no way to ask the player anything. A bargain needs both, and the unbind page is waiting for one.

28 Conversations with a shape

A topic table answers questions one at a time, and each answer is complete in itself. Most of what characters say in a game can be written that way. Some of it cannot. A bargain has an order: what is for sale, then the price, then yes or no. The person selling asks the last question, and waits.

This chapter adds three blocks that give a conversation an order. A **conversation** is a subject that a person takes forward a step at a time. An **exchange** is a question the person puts to the player, with the answers it accepts. An **initiative** block lets a person speak without being asked. It then returns to the topic table for three small things a row can do once conversations have a memory.

The Relighting uses all of it for one purpose. Tamsin Vey has the unbind page, and she will part with it for a promise.

```
create Tamsin Vey
  aka salvager, woman
  a person, proper, passive with announces-movement true
  pronouns she
  in the Refectory
  carries the unbind page
  states: free, bound
```

The page is no longer under the refectory candles. Her two states record whether the bargain has been struck.

28.1 A thread

```
define conversation the-bargain for Tamsin Vey
  about "page", "grille", "stair", "ward"
  beat:
    phrase tamsin-has-it
  beat:
    phrase tamsin-names-price
    then asks the-price
  on parting:
    phrase tamsin-parting
  on resuming:
    phrase tamsin-resuming
  conclusion:
    phrase tamsin-settled
end conversation
```

A conversation has a name and an owner. The Chord word for what it describes is a **thread**: one subject, carried forward in order.

The `about` line gives the subjects that belong to the thread, written as a topic row's are. The player does not need to know the thread exists. They ask about the page, or the grille, or the stair, and each such question moves the thread on by one `beat`.

```
> ask tamsin about the page
"I have a page that came out of a wall with a bracket," says
Tamsin Vey. "It was blank on Tuesday. It is not blank now. I am
told that is your sort of thing."
```

A beat's body is statements, as a topic row's is. The beats are taken in the order written. A beat may be held back with a condition, written `beat, when <condition>:`, and the thread then waits at that beat until the condition holds.

The thread is Tamsin's as much as the player's. If the player stays with her and lets a turn pass without a word, she takes the next beat herself. After the reply above, a `wait` is answered with her price, exactly as a second question about the page would be. A player who wanders off the

subject and comes back may find that a beat was spent while they were quiet.

28.1.1 Leaving a thread and coming back

The topic table is still there while a thread is under way. If the player asks about something else, the table answers, and the thread's `on parting` row is added.

```
> ask tamsin about the porter
"He pretends not to see me and I pretend to be quiet. It has
worked for eleven years."

"The page will keep," says Tamsin Vey. "So will I."
```

When the player comes back to one of the thread's subjects, the `on resuming` row answers. It is the person picking the subject up, and it does not use a beat.

```
> ask tamsin about the stair
"The page. Yes. I thought we would come back to it."

> ask tamsin about the page
"Here is my price. Lantern lit or Lantern dark, the brass in the
closed wings stays mine to take. Your word on it. Yes or no?"
```

Both rows are optional. A thread without them is simply left and taken up again without comment.

28.2 A question

The second beat ends with `then asks the-price`. That statement opens an exchange.

```
define exchange the-price for Tamsin Vey
  answer "yes", "agreed", "done", "you have my word":
    phrase tamsin-shakes
    change Tamsin Vey to bound
    Tamsin Vey gives the unbind page to Ansel
  answer "no", "never":
    phrase tamsin-shrugs
end exchange
```

An exchange holds answers only. The question itself is whatever the person said in the row that opened it. That keeps a question and the place it is asked together, and lets two rows open the same exchange with different words.

While an exchange is open the player may answer with the bare word.

```
> yes
She spits on her palm, thinks better of it, and holds out the page
instead. "Done is done. I have never gone back on one yet."

Tamsin Vey gives the unbind page to Ansel.

"That is the page dealt with," says Tamsin Vey, "one way or the
other."
```

Each `answer` row lists the replies it accepts, and the matching rule is the topic table's: the whole reply, never a guess. `say yes` and `answer yes` do the same as the bare word. A reply that no row accepts leaves the question standing.

```
> say maybe
Tamsin Vey waits for an answer to the question.
```

The player is not trapped by an open question. They may ask about another subject and the table answers, or walk away. The question is still open when they turn back to it.

`then asks` may be written `then invites` when the person is making an offer and not asking a question. The two open an exchange in the same way. Either is allowed in a thread's rows and in a topic row, and it opens an exchange belonging to the same person.

28.3 The conclusion

A thread ends with its `conclusion` row. Every thread has exactly one, and it runs once, after the last beat. Where the last beat opens an exchange, the conclusion runs when the exchange has been answered. You saw it close the transcript above.

Tamsin's conclusion runs whichever way the player answers. If the answer was no, the thread is over all the same.

```
> no
"Then I have a page and you have a grille." She does not sound
offended. "Ask me the price again when the stair matters more."

"That is the page dealt with," says Tamsin Vey, "one way or the
other."
```

A concluded thread no longer answers. Ask about the page now and Tamsin has nothing to add.

```
> ask tamsin about the page
Tamsin Vey gives that some thought, and gives it back.
```

That is the right behaviour for a thread, and it would be a dead end for the player, who has refused once and may think better of it at the foot of a sealed stair. So the bargain has a second way in.

28.4 A way back

A thread's name is something a condition can test.

```
define topics for Tamsin Vey
  about "price", "bargain":
    refuse when Tamsin Vey is bound: tamsin-already-dealt
    refuse when the-bargain is not concluded: tamsin-no-price-yet
    phrase tamsin-names-price
    then asks the-price
```

`the-bargain is concluded` is true once the thread's conclusion has run. This row is in the ordinary topic table. Before the thread has run its course it refuses, because Tamsin has not yet said there is anything to buy. Afterwards it states the price again and opens the same exchange.

```
> ask tamsin about the price
"A price for what? You haven't said what you want."

Tamsin Vey turns over a heap of door furniture on the end of a
table, and puts two hinges in her apron.
```

A refusal does not begin a conversation, so nothing holds Tamsin, and chapter 26's `rounds` takes its next step on the same turn. On the turn after that she walks out, as she did before this chapter.

```
> ask tamsin about the price
"Here is my price. Lantern lit or Lantern dark, the brass in the
closed wings stays mine to take. Your word on it. Yes or no?"

> agreed
She spits on her palm, thinks better of it, and holds out the page
instead. "Done is done. I have never gone back on one yet."

Tamsin Vey gives the unbind page to Ansel.
```

And when the bargain is struck, the same row closes the matter.

```
> ask tamsin about the price
"We have dealt, you and I. I don't sell a thing twice."
```

This is the shape to copy for any conversation that the player can get wrong: a thread that tells it in order once, and a plain topic row, gated on the thread being concluded, that lets the player return to its question.

28.5 What a table can remember

Three things in a topic row depend on what has been said before.

28.5.1 How many times

A row may answer differently the second time.

```
about "brass", "salvage", "hinges":
  phrase tamsin-on-brass when asked once
  phrase tamsin-on-brass-again when asked again
  phrase tamsin-on-brass-enough when asked many times
```

```
> ask tamsin about brass
"Door furniture, mostly. Lamp brackets. Nobody was using them."

> ask tamsin about brass
"Brass. It is yellow and it sells. I said."
```

`asked once`, `asked again` and `asked many times` are conditions that read how often this player has raised this row with this person. The third holds from the third asking onward.

A row whose phrases carry conditions must not leave two of them true at once. The compiler checks, and it will not accept a pair it cannot prove apart.

28.5.2 What has been discussed

A subject's name, written bare, can be tested with `was discussed`.

```
about "last night", "that night", "master":
  phrase hobb-on-last-night when dimming was discussed
```

Hobb will not speak of the night the Lantern was put out until the player has asked him about the Dimming. Before that, the row has nothing to say, and the stock answer is given.

```
> ask hobb about last night
Hobb gives that some thought, and gives it back.

> ask hobb about the dimming
He looks at the blue flame on the sill before he answers.

"Went out like a pipe," says Hobb. "No bang. The Master stood in
the court with his hand up for an hour and nothing came to it."

> ask hobb about last night
He takes his time, of which he has had a great deal.

Hobb is quiet for some time. "Somebody had to put the Lantern
out," he says. "It was guttering. The Master said a dead light was
worse than none. He didn't say who." He looks at his hands.
```

The subject is named by the first spelling of its row, without quotes. The test is kept for each pair of people, so what Ansel has discussed with Hobb is not what Ansel has discussed with Tamsin.

This one condition does a great deal. A table whose rows open one another in this way has an order that the player discovers by asking, and the author never has to draw it.

28.5.3 Sending a subject elsewhere

```
about "brackets", "door furniture":
  deflect to "brass"
```

`deflect to` answers this row with another row of the same table. It is for the subject that is really a different subject, and it keeps you from writing one answer twice.

28.6 Speaking first

Everything so far waits for the player. An `initiative` block gives a person lines to say unprompted.

```
define initiative for Tamsin Vey
  on silence, when Tamsin Vey is free:
    phrase tamsin-fills-silence
end initiative
```

A row begins with an occasion. `on silence` is the occasion when a conversation is open and the player has let a turn or two go by without saying anything. A thread comes first. While the bargain has a beat to give, a silent turn gets the beat, as the section on threads said. The silence row speaks in a conversation that has no thread under way, such as the one a question about brass opens.

```
> ask tamsin about brass
"Door furniture, mostly. Lamp brackets. Nobody was using them."

> wait
You wait. The Hall has had thirty years of practice and is better
at it than you are.

> wait
You wait. Somewhere a shutter knocks, twice, and thinks better of
it.

Tamsin Vey sorts hinges into sizes while she waits for you to say
something. There are more sizes than you would have thought.
```

An occasion may be narrowed with a comma and a `when`, as this one is. Once Tamsin is bound she has what she wanted, and she lets a silence be.

Use initiative sparingly. A character who speaks into every pause is tiring, and a single line that appears when the player hesitates in the middle of a bargain is worth more than a paragraph of description.

28.7 Build and try it

Copy `relighting.story` from the companion folder `chapter-28` over your own file. It is the story with this chapter's changes made, and it holds the phrases, which the chapter did not print in full. To make the changes by hand instead, work from this list and take the rest from that file: give Tamsin Vey the `carries` and `states` lines, take the `move the unbind page here` line out of the candles, and add her greetings, the conversation, the exchange and the initiative block, and the new rows in both topic tables.

```
sharpee compose relighting.story --check
sharpee play .
```

Go to the Refectory and ask Tamsin about the page twice, then answer her. Play it again and refuse, and find the way back.

The bargain can be struck before the bell has rung, and the page read as soon as the spark page has been. It does not open the stair early. The ward still refuses while the bell is unrung or tolling, as it has since chapter 24.

UNDER THE HOOD: SCENES · STORY IR `CONVERSATIONS`, `EXCHANGES`, `INITIATIVE`

The platform calls an open conversation between two people a scene. It opens when the player first addresses the person and closes when the player says goodbye or leaves. The compiled IR records a person's threads, exchanges and initiative rows beside the `topics`, `greetings` and `manner` of the last chapter, and at each turn of a scene the platform decides which of them answers: an open exchange first, then a live thread, then the topic table. The standard action behind a bare `yes` is `answering`, and the platform offers a bare reply to an open exchange before it tries to read the reply as a command, which is why `yes` works with no verb.

FROM ANOTHER SYSTEM: CONVERSATION NODES

Writing with Inform describes no built-in structure for a conversation that proceeds in order. It shows tables with topic columns, and a variable that remembers the current question. TADS 3 has a structure very close to Chord's: a `ConvNode` is a point in a conversation, and `YesTopic` and `NoTopic` objects placed in it answer yes and no while the character waits for a reply. A Chord exchange corresponds to a `ConvNode`, and its `answer` rows to the topic objects inside one. A TADS 3 character begins a conversation unprompted through an `InitiateTopic` or an agenda item.

28.8 What you have so far

The fourth page has a price. Tamsin Vey tells the player what she has, names what she wants for it, asks, and holds the question open until it is

answered. A refusal can be mended. Hobb's table has a row that opens only after another has been heard.

Both of them now say a good deal, and neither of them yet feels anything. Tamsin's promise is a state with two values, and Hobb's fondness for a returning student exists only in the prose. The next two chapters give people an inside.

29 Inner lives

Hobb and Tamsin Vey can now say a great deal, and what they say depends on things: whether the bell is tolling, whether a bargain is struck, what has been asked before. All of that is the state of the world. None of it is the state of the person.

A person in Chord has an inside as well, and the language calls it the **character model**. You have already written one line of it, the `mood calm` that chapter 27 needed. This chapter covers the three parts of the model whose effects you can see and use in a story: a mood, feelings toward other people, and things known.

Each works the same way. A line in the person's block says how they begin. A statement changes it. A condition reads it, and anything that takes a condition can then depend on it: a topic row, a manner row, a clause, a goal.

29.1 Mood

```
create Hobb
  a person, proper
  pronouns he
  in the Lodge
  carries the hush page
  mood calm
```

The `mood` line gives the mood a person starts in. The platform has thirteen mood words: `calm`, `content`, `cheerful`, `nervous`, `anxious`, `panicked`, `angry`, `furious`, `sad`, `grieving`, `suspicious`, `confused` and `resigned`.

A mood is read with `is`, as a state is.

```
define manner for Hobb
  when Hobb is calm:
    beat "He turns the kettle a quarter round on its hook, to no
    ↪ purpose."
    beat "He looks at the blue flame on the sill before he
    ↪ answers."
    beat "He takes his time, of which he has had a great deal."
  when Hobb is rattled:
    beat "He takes one hand off one ear to hear you, and puts it
    ↪ back."
end manner
```

In chapter 27 these two rows asked about the bell. They now ask about Hobb, which is what a manner should be about. The bell is the reason, and the reason is written once, somewhere else.

29.1.1 A mood of your own

`rattled` is not one of the thirteen. A story may define a mood by saying which platform mood it resembles.

```
define mood rattled like nervous, but darker
```

The line names the new mood and the platform mood it is `like`. It may end with `, but` and one of four words that shift it: `restless`, `stiller`, `darker` or `brighter`. A defined mood is used wherever a platform mood is: on a `mood` line, after `change mood to`, and in a condition.

29.1.2 Changing a mood

```
on every turn while the tower bell is tolling
  change mood to rattled
end on
```

`change mood to` has no person in it. It changes the mood of the person whose block or table the statement stands in. This clause stands in Hobb's own block, so it is Hobb who is rattled. Written in one of Tamsin's topic rows it would change Tamsin's mood. Always write it in a person's block or in one of that person's tables, where it has someone to belong to.

```
> ask hobb about the bell
He takes one hand off one ear to hear you, and puts it back.

"It'll go on till it's told," Hobb shouts. "There's a word for the
telling. I kept it by, in a newspaper, in case."

Hobb gives the hush page to Ansel.

The bell tolls again, and the rooks go up off the roof.
```

A mood does not stay changed. Left alone, a person settles back to the mood they began in over a few turns. That is why Hobb's clause runs on every turn the bell tolls: it holds him rattled for as long as there is something to be rattled by. Hush the bell and he settles. For the first few turns afterwards his answers come with no sentence of manner at all, and then he is calm again and turning his kettle.

This is worth designing around. A mood is the right tool for how someone is taking the present moment. For a change that should last, use a feeling, or a state.

29.2 Feelings

A person may feel something toward another person, or toward any thing or place in the story.

```
feels wary of Ansel
```

Hobb's block has that line, and so has Tamsin Vey's, with a `mood suspicious` above it.

There are eight words, and they run from worst to best: `despises`, `hates`, `dislikes`, `wary of`, `neutral`, `likes`, `trusts` and `devoted to`. Two of them end in a preposition. After the other six, write `toward` before the name: `feels trusts toward Ansel`.

A feeling is read with the same words.

```
about "last night", "that night", "master":
  refuse when Hobb feels wary of Ansel: hobb-not-to-strangers
  phrase hobb-on-last-night when dimming was discussed
```

Hobb's best story now has two locks on it. The player must have asked about the Dimming, as before, and Hobb must not be wary of the person asking.

```
> ask hobb about last night
Hobb looks at you for a while, narrowly, as a man looks at small
print. "That's not a thing I tell to people I can't see," he says.
```

The refusal is the hint. Hobb has been looking for his spectacles since the spring, and chapter 24 made them something Ansel can mend.

29.2.1 Changing a feeling

```
on every turn while Hobb wears the spectacles, once
  change feeling toward Ansel to trusts
end on
```

This clause is in Hobb's block too.

`change feeling toward <someone> to <word>` follows the rule of `change mood to`: the feeling changed is that of the person whose block the statement stands in. Unlike a mood, a feeling stays where it is put.

```
> give spectacles to hobb
You give the spectacles to Hobb.

Hobb hooks them over his ears and looks at you properly for the
first time. "Ansel," he says, after a while. "You were always
late."

> talk to hobb
He turns the kettle a quarter round on its hook, to no purpose.

"Still here," says Hobb, of one of you.

> ask hobb about last night
He looks at the blue flame on the sill before he answers.

Hobb is quiet for some time. "Somebody had to put the Lantern
out," he says. "It was guttering. The Master said a dead light was
worse than none. He didn't say who." He looks at his hands.
```

Tamsin's feeling changes in the row that strikes the bargain.

```
define exchange the-price for Tamsin Vey
  answer "yes", "agreed", "done", "you have my word":
    phrase tamsin-shakes
    change Tamsin Vey to bound
    change feeling toward Ansel to likes
    Tamsin Vey gives the unbind page to Ansel
```

And one of her topic rows has two answers, chosen by it.

```
about "magic", "craft":
  phrase tamsin-on-magic when Tamsin Vey feels wary of Ansel
  phrase tamsin-on-magic-warmer when Tamsin Vey feels likes toward
  ↪ Ansel
```

```
> ask tamsin about magic
"My mother kept a stall by this gate. The day it went out, so did
the custom. Now you tell me it's coming back, and I am to be
glad." She is not, particularly.
```

```
> ask tamsin about magic
"I'll tell you what I never told the old man. The night it went
out, my mother's scales would not weigh true, and she laughed. She
said it was the first honest day's trade she ever did." Tamsin Vey
looks at the page in your hand. "Light your lamp. I would like to
see if she was right."
```

The two phrases in that row carry conditions, and chapter 28 said that a row's conditions must not overlap. Two different feelings toward the same person never do, and the compiler knows it.

A state would have done this job. Tamsin already has `free` and `bound`, and the second phrase could have asked whether she is bound. The feeling is better for two reasons. It says what you mean, so that the row still reads correctly when a second way of winning her round is added. And it is a scale and not a switch: a story can move a person from `wary of` to `neutral` to `likes` in three steps, and write a row for each.

29.3 What a person knows

```
knows the-last-night, witnessed, certain
```

A `knows` line in a person's block says that the person holds a piece of knowledge when the story starts. The knowledge is a name of your choosing, a single hyphenated word, and it stands for the thing known. It does not need to be declared anywhere else.

Two more words follow it. The first is how the person came by it, and is required: `witnessed`, `told`, `inferred`, `assumed` or `hallucinated`. The second is how sure they are: `uncertain`, `suspects`, `believes` or `certain`.

A condition reads it with `knows`.

```
phrase hobb-keeps-it when Hobb knows the-last-night
```

By itself that is a flag, and a state would serve. The value of naming knowledge is that the names are shared. Two people can know the same thing, and a row in one person's table can ask what another person knows.

29.3.1 Telling one another

Knowledge also moves. A `spreads` line says what a person passes on, and to whom.

```
create Nell
  a person, proper
  in the Yard
  mood calm
  knows theft, witnessed, certain
  spreads theft to anyone

  Nell.

create Tom
  a person, proper
  in the Yard
  mood calm
  spreads nothing

  Tom.
```

`spreads` is followed by the things spread, then `to` and an audience: `anyone`, `trusted` or `allied`. Several things are joined with `and`. When two people who each have a `spreads` line are in a room together, the platform has each tell the other what their line allows. Tom's line is `spreads nothing`. He passes nothing on, and the line is what makes him someone who can be told.

```
> wait
Time passes...

You hear: "Nell mentions something to Tom."
```

From that turn `Tom knows theft` is true, and anything gated on it opens. The player, standing in the Yard, is told only that something was said. In another room the player would be told nothing, and Tom would know all the same.

That is rumour, in two lines a person. A fact witnessed in the kitchen can reach the drawing room by way of whoever walks between them, and what the people there will say to the player changes as it arrives.

29.4 Build and try it

Copy `relighting.story` from the companion folder `chapter-29` over your own file. It is the story with this chapter's changes made, and it holds the phrases, which the chapter did not print in full. To make the changes by hand instead, work from this list and take the rest from that file. Add the `define mood` line at the top of the story. Give Hobb his `feels` and `knows` lines and the two `on every turn` clauses, and change his manner rows to read his mood. Give Tamsin Vey her `mood` and `feels` lines, add the `change feeling` line to her exchange, and split her row about magic in two.

```
sharpee compose relighting.story --check
sharpee play .
```

Ask Hobb about the Dimming and then about last night. Then find his spectacles, give them to him as they are, and ask again. Do this before you read the spark page. Hobb trusts whoever puts the spectacles on his nose, mended or not, and while the Hall is quiet he answers as the blocks above show. Once the bell has begun he is rattled, and he answers the same questions with one hand over an ear.

UNDER THE HOOD: THE CHARACTER MODEL · STORY IR `CHARACTER`

Every line of this chapter is recorded under one key of the person's entity in the compiled IR, `character`. It holds `mood`, a `feels` list with one entry for each person or thing felt about, and a `knows` list in which each entry carries its source and its confidence. The goals of chapter 26 are in the same place. The platform builds one model for each person from that key when the story loads, and it is the model's own clock that brings a mood back to where it began.

FROM ANOTHER SYSTEM: FEELINGS AS RELATIONS

Neither manual describes a built-in model of mood or feeling. In Inform 10 the usual tool is a relation: a sentence declares that it "relates" people to people, another gives it a verb, and after that a sentence with the verb is a condition like any other. The TADS 3 documentation suggests a property on the actor, tested where a response should depend on the character's frame of mind. Chord's `feels` is a relation of the Inform kind that the language already knows about, with a fixed scale of eight values where Inform's would be yours to define.

29.5 What you have so far

Hobb is rattled while the bell tolls and calm when it stops, and his manner follows his mood. He is wary of a stranger and trusts a returned student, and the difference is a pair of spectacles. Tamsin Vey warms to someone who has kept a bargain with her, and says so in her own way.

30 Principles

In the last chapter Hobb kept one story back from a stranger. The row that kept it back said so in as many words.

```
about "last night", "that night", "master":
  refuse when Hobb feels wary of Ansel: hobb-not-to-strangers
  phrase hobb-on-last-night when dimming was discussed
```

That works, and it is the author deciding. The row knows that this question is delicate, knows which feeling matters, and has a refusal written for the case. A character with twenty things to keep quiet about needs twenty such lines, and each is a place to get the rule slightly different.

Chord lets you say the rule once, about the person. Hobb does not betray a confidence. What he was told in confidence is marked as such where he knows it. Whether he tells is then worked out by the platform each time he is asked, from what he holds to and how he feels about the person asking. This chapter is about the lines that say what a person holds to.

30.1 Something told in confidence

A `knows` line may end with one more word.

```
knows night, told, certain, confided
```

`confided` says the knowledge was given in trust. It changes nothing by itself. A person with no principles tells a confided thing as readily as any other.

The name matters here in a way it did not in the last chapter. The platform connects a fact to a topic row by name: the fact is `night`, and the row that speaks of it must have `night` as one of its spellings.

```
about "night", "last night", "that night", "master":
  phrase hobb-on-last-night when dimming was discussed
```

The row has lost its `refuse` line. It now says only what Hobb would say, and when the subject may come up at all. Whether he says it is no longer the row's business.

30.2 A thing a person will not do

```
never betrays a confidence
```

A `never` line goes in the person's block. It names an act the person holds it wrong to commit. There are seven: `betrays a confidence`, `lies`, `harms`, `steals`, `breaks a promise`, `abandons` and `trespasses`. Write the act in the third person, as it reads after the person's name.

With that line, a question that would have Hobb reveal a confided thing meets his principle, and the principle wins.

```
> ask hobb about last night
Hobb gives that some thought, and gives it back.
```

Nothing in the row refused. The platform judged that answering would be the forbidden act, and withheld the row. What the player reads is the story's line for a subject a person has nothing to say about, the one The Relighting overrode in chapter 27. From outside, a kept confidence and plain ignorance look the same, which is the point of keeping one.

A principle may have an exception, written after a comma.

```
never betrays a confidence, except Ansel
```

The exception names a person, or a kind of person such as `a servant`, or `anyone`. A person with this line tells Ansel and no one else.

30.3 What decides between principles

One rule and no exceptions gives a character who never tells. People are more interesting when two things they hold to pull against each other, and Chord has a word for how a person settles that.

The platform recognises five forces that can move a person: `fear`, `desire`, `duty`, `honor` and `love`. A principle is a duty. A feeling of trust or devotion toward someone is love. When two forces disagree about an act, the stronger one wins, unless the person has a `temperament` line that ranks them.

```
temperament love over duty
```

A temperament is one or more pairs, each `<force> over <force>`, joined with `and`. This one says that where love and duty collide, love decides.

Here is Hobb in full.

```
create Hobb
  a person, proper
  pronouns he
  in the Lodge
  carries the hush page
  mood calm
  feels wary of Ansel
  knows night, told, certain, confided
  never betrays a confidence
  temperament love over duty
```

While he is wary of Ansel there is no love in the question, and duty is unopposed. He keeps the confidence, as above. Then the spectacles are given, mended or not, and the clause from the last chapter changes his feeling to `trusts`.

```
> give spectacles to hobb
You give the spectacles to Hobb.

Hobb hooks them over his ears and looks at you properly for the
first time. "Ansel," he says, after a while. "You were always
late."

> ask hobb about last night
He takes his time, of which he has had a great deal.

Hobb is quiet for some time. "Somebody had to put the Lantern
out," he says. "It was guttering. The Master said a dead light was
worse than none. He didn't say who." He looks at his hands.
```

Nothing in the row changed between those two answers. The row never mentioned trust. Hobb told because he is a man who puts a person he trusts above a rule he was given, and the story said so in one line of his block.

The puzzle is the one the last chapter had, and the player solves it the same way. What has changed is where the rule lives. Give Hobb a second confidence tomorrow, with a row of its own, and it is kept and given up on exactly the same terms, with nothing more to write.

30.4 Honesty

A second line of the same family pulls the other way.

```
answers honestly
```

A person who `answers honestly` holds it a duty to answer what they are asked. Alone, with no `never` line, it changes little: such a person tells a confided thing to whoever asks.

Its use is in the collision. A person with both lines has two duties, one forbidding the answer and one requiring it, and a temperament ranks

forces and not two duties against each other. The platform's answer to that is the human one. The person evades.

```
create Cy
  a person, proper
  in the Yard
  mood calm
  knows secret, told, certain, confided
  never betrays a confidence
  answers honestly

  Cy.
```

```
> ask cy about secret
Cy says, "I don't know anything about that."
```

And a feeling can close a mouth as well as open one. A person who `answers honestly` and has no rule against telling still keeps a confidence from someone they hate.

30.5 What the lines add up to

Six people were each told the same secret in confidence, and each was asked for it by the player. This is what they did.

The person's lines	Asked, they
no principle	tell
`never betrays a confidence`	keep it
`never betrays a confidence`, `except` the asker	tell
`never betrays a confidence`, `temperament love over duty`, and devoted to the asker	tell
`answers honestly`	tell
	keep it

The person's lines	Asked, they
`answers honestly`, and hates the asker	
`never betrays a confidence` and `answers honestly`	evade

Read down the table and you have a small cast: the gossip, the vault, the confidant, the friend, the honest witness, the enemy, and the person in a bind. Each is two or three lines.

Principles of this kind are worth their cost when a story has several people and several secrets. For one secret and one keeper, the last chapter's `refuse when` is shorter and says what you mean. The gain comes with the second secret, and with the first time you change a character's mind about someone and find that everything they would and would not tell has moved with it.

30.6 Build and try it

Copy `relighting.story` from the companion folder `chapter-30` over your own file. It is the story with this chapter's changes made, and it holds the row about Hobb's spectacles, which the chapter did not print. To make the changes by hand instead, work from this list and take the rest from that file. Change Hobb's `knows` line and add the two lines beneath it. Take the `refuse` line out of his row about the last night and give the row the spelling `night`. Add a row about his spectacles, so that the player has something to go on.

```
sharpee compose relighting.story --check
sharpee play .
```

Ask Hobb about the Dimming, then about the last night, then about his spectacles.

```
> ask hobb about spectacles
He turns the kettle a quarter round on its hook, to no purpose.

"Lost them in the spring," says Hobb. "Since then a face is a
face. There are things I was told that I don't tell to a face."
```

UNDER THE HOOD: THE ARBITER · STORY IR

`CHARACTER.PRINCIPLES`, `CHARACTER.TEMPERAMENTS`

The compiled IR records a person's `never` lines under `character.principles` and the temperament under `character.temperaments`, beside the `knows` entry that carries the `confided` mark. When a topic row is about to answer, the platform checks whether any spelling of the row names a fact its owner holds in confidence. If one does, it weighs the forces for and against answering: each principle, the person's feeling toward the asker, and the temperament's ranking. A verdict to refuse or to evade withholds the row, and nothing in the row runs.

FROM ANOTHER SYSTEM: KNOWLEDGE AND RETICENCE

Neither manual describes a model of what a character will and will not reveal. The TADS 3 documentation says of its conversation system that the library does not use any sort of knowledge model for non-player characters, and suggests an `isActive` test on a topic to model a character's frame of mind. That is the approach of this chapter's first example, the `refuse when` on a row. Chord's principles are a layer above it, in which the test is derived from lines about the person.

30.7 What you have so far

Hobb keeps what he was told in confidence, from anyone he does not trust, because of who he is and not because a row says so. The game plays as it did. Its porter has become someone who could keep a second secret without being taught how.

31 More than one protagonist

Every chapter so far has had one player character. Ansel is `playable`, the start block says `change the player to Ansel`, and the word `player` in a thousand lines of clauses has meant Ansel.

It did not have to. In Chord the player is a role, and a role can be handed from one person to another while the story runs. This chapter gives The Relighting a second protagonist for one scene. When the bell begins to toll, the story leaves Ansel in the Library and goes to the Lodge, where an old man has just opened his eyes.

31.1 Two people who can be played

```
create Hobb
  a person, proper
  playable
  pronouns he
  in the Lodge
  states: dozing, roused, settled
  mood calm
```

`playable` is the line Ansel has had since chapter 3. Any number of people may have it. It says that the role may be given to this person, and it says nothing about when.

The start block still chooses who begins.

```
before the game starts
  change the player to Ansel
end before
```

A playable person who is not the player is an ordinary person. Hobb stands in his doorway, answers questions, and shuts his press, exactly as before.

Handing the role to someone who is not `playable` is an error when you build.

```
probe.story:23:5 error [analysis.player-target-not-playable]
`nell` is not `playable` — add `playable` to its create block to
let the player role move to it.
```

31.2 Handing the role over

The statement is the one from the start block. Written anywhere else, it moves the role during play.

```
when waking expires
  change the tower bell to tolling
  change Hobb to roused
  phrase hobb-wakes
  change the player to Hobb
end when
```

The waking timer from chapter 19 expires on the turn the bell begins to toll. That clause, in the story header, now does three more things. It puts Hobb in a state of his own, prints a passage, and hands him the role.

```
> light lamp with taper
You have no match, and you did not walk here for three days to use
one. There is a word for this.

In the lodge, an old man who was not asleep opens his eyes.

You are Hobb, and that is the bell. It has not rung since the
night you stopped it. There was a word for stopping it, and you
wrote the word down, because you knew you would not keep it in
your head, and you put the paper somewhere safe.

The tower bell tolls.
```

The platform does not announce the change, and it does not describe the new room. The player's next prompt simply belongs to someone else. So the prose at the moment of the change has work to do: it must say who the player now is and where, and it should give them a reason to do something. That is what `hobb-wakes` is for.

From the next command on, everything the player types is Hobb's.

```
> look
The Porter's Lodge
The porter's lodge is one small room built into the thickness of
the gate arch. A chair, a kettle and thirty years of newspapers
have filled most of it. On the sill a lamp is burning with a
steady blue flame, and an oak press hangs on the wall beside it.
The door to the court is north.

You can see a kettle here.

On the chair you see a seed cake and a cup.

The bell tolls again, and the rooks go up off the roof.

> inventory
Your hands are empty.

The bell tolls. You have stopped counting.
```

The platform has three ways of saying that an inventory is empty and picks one each time, so yours may speak of pockets, or of carrying nothing at all.

Each person keeps what is theirs. Hobb is in the Lodge because that is where Hobb was. His hands are empty because Ansel has the lamp, the key and the pages, and Ansel is in the Library with them. The score and the story's states belong to the story, and they carry across.

31.3 The player is whoever has the role

Open the story and search for `the player`. Every clause that says it was written with Ansel in mind, and every one of them now applies to Hobb as well, for as long as he has the role. The word is resolved when a clause fires, and it means whoever the player is at that moment.

Most of the time that is exactly right. The grimoire's clauses, the workings and the blocked stair treat any player alike. Now and then it is wrong, and the kettle is the first case.

```
on the player taking while Hobb is here
  refuse kettle-is-hobbs
end on
```

Chapter 14 wrote this so that Ansel could not walk off with Hobb's kettle. Played as Hobb, it would have Hobb refuse himself, in the third person. The condition needs a second half.

```
on the player taking while Hobb is here and Hobb is not roused
  refuse kettle-is-hobbs
end on
```

```
> take kettle
You take it.

The bell tolls again, and the rooks go up off the roof.
```

There is no condition that asks who the player is. The working form is the one used here: give the second protagonist a state that is true while they are being played, and test the state. Hobb is `roused` for exactly the length of his scene, and three things in the story lean on it.

31.3.1 What a room says about someone

The second case is the room. Since chapter 21 the Lodge has carried a sentence about Hobb standing in the doorway, written as his own `phrase present`. A player who is Hobb should not read it.

A sentence that belongs in a description only while something is true is a gated phrase, from chapter 20.

```
define phrase hobb-in-doorway while Hobb is not roused
  Hobb is in the doorway with his back to you, watching the court
  ↪ as if
  it might be up to something.
end phrase
```

The Lodge's description ends with the marker `{hobb-in-doorway}`, and Hobb's own block no longer has a `phrase present`. The transcript above shows the result: the room without its porter, as its porter sees it.

The same care is owed to a playable person's description. Hobb's says he "was old when you were a student", which is Ansel's view of him. If the player may examine themselves as Hobb, write the description so that it reads from either side, or write a clause for the other case.

31.3.2 Keeping a scene where it belongs

A second protagonist is free to walk out of the scene you wrote for them. Sometimes that is the point. Here it is not, and one line holds him.

```
create the Lodge, a room
  in the Grounds
  north to the Gate Court
  north is blocked while Hobb is roused: hobb-stays-put
```

```
> north
Not with that going on. And not until you have found the thing.

The tower bell tolls.
```

This is the blocked exit of chapter 12, gated on the same state. Ansel passes through that door all day.

31.4 Handing it back

The scene ends when Hobb finds what he hid.

```
create the newspapers
  aka papers
  a thing, scenery, plural
  in the Lodge

  The top one is from last week. The bottom one reports the
  ↪ Dimming, in
  the small type newspapers use for things they do not yet
  ↪ believe.

  after the player examining
    move the hush page to Hobb when Hobb is roused
    phrase hobb-finds-page when Hobb is roused
    change the player to Ansel when Hobb is roused
    phrase ansel-again when Hobb is roused
    change Hobb to settled when Hobb is roused
  end after
```

Each statement has the same `when`. For Ansel, who may examine the newspapers at any time, the clause does nothing. For Hobb in his scene it does five things in order.

```
> examine newspapers
The top one is from last week. The bottom one reports the Dimming,
in the small type newspapers use for things they do not yet
believe.

You lift the pile, which your back will mention later, and take
out the bottom paper. The leaf is still folded inside it where you
put it the morning after. You keep it in your hand for whoever
comes asking. They will come asking. They always wanted telling
twice.

You are Ansel, and wherever you are standing, the bell is louder
there.

The tower bell tolls.

> look
Library
The reading room of Lanternhall, with its desks in rows and its
galleries of shelves. The stacks run back to the north, out of the
light. The cloister is east.

You can see a wisp here.

On the reading desk you see a grimoire.

The tower bell tolls.
```

The hush page is no longer something Hobb carries from the start. It was in the newspapers, and the player put it in his hand. The topic row from chapter 27 does the rest when Ansel comes asking: `Hobb gives the hush page to Ansel when Hobb has the hush page`.

Notice the order of the last three statements. The state changes last, because every `when` above it reads the state. Had `change Hobb to settled` come first, nothing after it would have run.

Ansel is where the player left them, and the wisp has waited. A follower follows the player, but it follows a person: the wisp was brought on beside Ansel's lamp, and it stays with Ansel.

31.5 When to do this

A second protagonist is a large effect, and it should buy something. Here it buys three things. The player learns where the third working is without being told. Hobb, who has been a voice in a doorway since chapter 6, gets a minute of an inner life that no topic table could have given him. And the bell, which until now was a nuisance on every turn, becomes something that happened to somebody.

It costs a review of every `the player` in the story. Make that review a habit the moment a second person is `playable`.

31.6 Build and try it

Copy `relighting.story` from the companion folder `chapter-31` over your own file. It is the story with this chapter's changes made, and it holds the phrases, which the chapter did not print in full. To make the changes by hand instead, work from this list and take the rest from that file. Give Hobb `playable` and his three states, and take away his `carries` line and his `phrase present`. Add the three statements to the header's `when waking expires`, the blocked line and the marker to the Lodge, the gated phrase, the clause on the newspapers, and the second half of the kettle's condition.

```
sharpee compose relighting.story --check
sharpee play .
```

Read the spark page and play four more turns. The timer speaks on the first three, and it expires on the fourth, when the bell begins to toll. When you are Hobb, try the door, take the kettle because you can, and then look through the newspapers.

UNDER THE HOOD: A ROLE, NOT A PERSON · STORY IR `ISPLAYABLE`, `STARTBLOCK`

In the compiled IR each person has an `isPlayable` flag, and the start block is recorded under `startBlock`.
`change the player to` is one statement kind wherever it appears. The platform reads it one way inside the start block, as the first assignment of the role, and another way on any later turn, as a request to move the role when the turn ends. That is why the rest of the clause still runs for the outgoing player: the role moves after the statements, not between them. The world keeps one flag saying which actor holds the role, and every "you" in a standard message is worked out from it.

FROM ANOTHER SYSTEM: CHANGING THE PLAYER

Inform 10 has the same idea in one phrase: "now the player is Hobb" switches the perspective, so that Hobb is the one the human player controls. TADS 3 names the starting player character in the game's definition, as `initialPlayerChar`.

31.7 What you have so far

That is Part V. Lanternhall has four people in it. One follows a lamp, one walks her rounds and drives a bargain, and one keeps a door, tells what he knows to people he trusts, and for one scene is the player. Ansel can ask, tell, answer and consult.

The Hall itself is still a simple machine. Its Lantern lights when a lens is in it and a word is said. Part VI is about larger mechanisms: things with

several states that drive one another, things that carry people from place to place, and places that are not rooms at all.

PART VI. LARGER MACHINERY

32 Machines

A thing with states, from chapter 13, changes when a clause on that thing says `change`. That is the right tool for a page that is faded, legible or learned, where one action moves one thing one step.

Some mechanisms are not like that. A lock with a sequence, a lift with floors, a ritual in four parts: each has a state of its own that belongs to no single thing, and what the player does to one part depends on where the whole has got to. You can build these from states and clauses, and by the third state the logic is spread across six blocks and nobody can see it.

A **machine** gathers that logic into one block. It lists the states, and under each state the things that move the machine out of it. This chapter builds one: the shutters of the Great Lantern, which have to be cranked open before a flame in the Lantern can be seen by anybody.

32.1 The parts

A machine is not a thing. It has no description and the player cannot refer to it. It watches things, so the things come first.

```
create the shutter
  aka shutters, leaves
  a thing, scenery
  in the Lantern Gallery
  states: closed-round, drawn

  A shutter of brass leaves that closes round the Lantern like a
  ↪ bud,
  worked by a crank on the drum.

  phrase detail while the shutter is closed-round:
    The leaves are shut, and green where they overlap.

  phrase detail while the shutter is drawn:
    The leaves are folded back all the way round.

create the shutter crank
  aka crank, handle
  a thing, scenery
  in the Lantern Gallery

  An iron crank on the side of the drum, with a wooden grip worn
  ↪ to the
  shape of a hand that is not yours.
```

The shutter has two states because the player can see two conditions of it. The crank has no states and no clauses. Everything it does, the machine will say.

32.2 The machine

```
define machine shutter-gear
  role crank is the shutter crank
  starts shut

  state shut
    when turning the crank: half

  state half
    on enter
      phrase shutters-shift
    end on
    when turning the crank while the lens is in the Great Lantern:
    ↪ drawn
    when pushing the crank: shut
    on exit
      phrase gear-ticks
    end on

  state drawn, terminal
    on enter
      change the shutter to drawn
      phrase shutters-open
    end on
end machine
```

`define machine` is a top-level block with a name. Machines need no `use` line. Read the body in three parts.

A `role` line gives a short name to a thing the machine watches. `role crank is the shutter crank` lets the rest of the block say `the crank`. A machine may have several roles, and it may also name things directly.

The `starts` line names the state the machine begins in.

Each `state` block is one state, and its lines say what happens there. These states are the machine's own. They are separate from the shutter's `closed-round` and `drawn`, and a machine may have more states than anything the player can see. This one has three where the shutter has

two, because the gear has a half-way notch at which the leaves have not really opened.

32.3 Moving between states

A `when` line is a transition. It names a trigger, then a colon, then the state to go to.

```
when turning the crank: half
```

The trigger here is an action on a role: the gerund of chapter 14 and the thing it is done to. While the machine is in the state `shut`, turning the crank moves it to `half`. In any other state this line is not in force.

```
> turn crank
You turn the shutter crank.

Something inside the drum takes up the slack. The leaves of the
shutter part by a hand's breadth all the way round, and stop.
```

The first line of the reply is the platform's. A thing that a machine watches for turning becomes a thing that can be turned, and the standard action reports it. The crank needed no trait word.

A transition may carry a guard, written with `while`.

```
when turning the crank while the lens is in the Great Lantern:
↪ drawn
```

The gear is interlocked with the lens seat. With the seat empty, the crank still goes round, and the machine stays where it is.

```
> turn crank
You turn the shutter crank.
```

A state may have several ways out. From `half` the machine also goes back.

```
when pushing the crank: shut
```

```
> push crank
You push the shutter crank.

The pawl ticks over one tooth.
```

A trigger may also be a plain condition, with no action in it. `when the cistern is running: flooded` moves a machine as soon as the condition becomes true, whatever the player was doing. The platform looks at such a line on every turn while the machine is in that state.

32.4 Entering and leaving

A state may have an `on enter` block and an `on exit` block. Each holds statements, as a clause does, and each runs when the machine arrives in the state or leaves it, by whichever transition.

The state `half` has both. Its `on enter` printed the leaves parting, and its `on exit` is the pawl. That is why the pawl ticked when the crank was pushed: the machine left `half`. It ticks again on the way to `drawn`.

```
> turn crank
You turn the shutter crank.

The pawl ticks over one tooth.

The crank runs free, and the shutters fold back all the way round,
like a flower that has made up its mind.
```

Put prose and effects in `on enter` and `on exit`, and keep the `when` lines for the map of the machine. The benefit comes when a state gains a

second way in. Its consequences are written once, on the state, and both ways in get them.

Inside a machine there is no `it`. A machine belongs to the story and not to a thing, so every statement names what it acts on.

32.5 Telling the world

The player cannot see a machine, and a clause on a thing does not ask a machine what state it is in. A machine makes itself felt by changing things.

```
state drawn, terminal
  on enter
    change the shutter to drawn
    phrase shutters-open
  end on
```

Arriving at `drawn` writes the shutter's own state. Everything else in the story reads the shutter, in the ordinary way. The description's `phrase detail` lines read it. So does the Lantern.

```
define trait lens-seat
  on the player sparking
    the lens must be in the Great Lantern: lantern-empty
    the shutter must be drawn: lantern-shuttered
  end on
end trait
```

```
> spark lantern
The shutters are closed round it. A flame in there would light the
inside of a brass bud, and nothing else.
```

This is the division of labour to aim for. The machine owns the sequence. A thing owns a state the player can see. The rest of the story depends on the thing and never needs to know a machine exists.

`terminal` after a state's name says the machine stops there. Once the shutters are drawn the crank turns freely and nothing more happens.

32.6 Build and try it

Copy `relighting.story` from the companion folder `chapter-32` over your own file. It is the story with this chapter's changes made, and it holds the phrases, which the chapter did not print. This is the first chapter of Part VI, and the copy's `story-version` is `0.6.0`. To make the changes by hand instead, work from this list and take the rest from that file: add the shutter and the crank to the Lantern Gallery, the machine near the top of the story, the second `must` line to the `lens-seat` trait, and four phrases.

```
sharpee compose relighting.story --check
sharpee play .
```

The last room of the winning walk now takes a little longer. Turn the crank once before the lens is seated and once after, and try to spark the Lantern in between.

> **UNDER THE HOOD: THE STATE MACHINE · STORY IR `MACHINES`**
>
> The compiled IR has a top-level `machines` list. Each machine carries its roles, its starting state, and for each state its transitions and its enter and exit bodies. A transition's trigger is recorded as one of a few kinds: an action with the entity it is done to, or a condition. The platform's state-machine plugin holds the current state of each machine, moves it when a trigger in that state fires, and runs the exit body of the state it leaves and the enter body of the state it reaches.

FROM ANOTHER SYSTEM: STATES AND SCENES

Inform 10 has no construct called a machine. Its nearest tools are a kind of value for the states ("The shutter-gear is a kind of value. The shutter-gears are shut, half and drawn") with rules that test and set it, and scenes, which begin and end on conditions and can run rules when they do. Chord's block differs in putting every transition of a mechanism in one place, grouped by the state it leaves.

32.7 What you have so far

The Great Lantern is now a mechanism in three parts: a lens in a seat, shutters drawn back by a crank, and a word. The interlock means the lens must be seated before the shutters will open fully, and the shutters must be open before the word will take. A player who arrives on the roof with a mended lens has perhaps six commands still to find.

The lens itself is as wide as a cartwheel, and so far Ansel has carried it up the tower stair under one arm. The next chapter has an opinion about that.

33 Vehicles

A chair you can sit in is `enterable`, and has been since chapter 7. A vehicle is an enterable thing that goes somewhere, and takes what is in it along.

Chord's vehicles are declared, not scripted. You say what kind of movement a vehicle has, what drives it, and where it may go, and the platform moves it, reports its arrivals, and carries its passengers. This chapter builds the simplest useful kind for The Relighting, a hoist that goes where its load sends it, and then shows the other kinds in short examples.

33.1 A problem for a vehicle

The lens of the Great Lantern is a disc of glass as wide as a cartwheel. Since chapter 9 Ansel has carried it up a winding tower stair without comment. One line ends that.

```
create the Bell Stair, a room
  in the Hall
  up to the Lantern Gallery
  up is blocked while the iron grille is standing: stair-sealed
  up is blocked while Ansel has the lens: lens-too-wide
```

A room may have two blocked lines for one direction, each with its own condition and its own phrase. While the grille stands, the player still reads the grille's phrase. With the grille lifted and the lens in hand:

```
> up
The lens is as wide as a cartwheel and the stair is as wide as a
student. One of you is not going up this way.
```

The lens needs another way to the roof. A college with a library on the ground floor and readers at the top of a tower has one already.

33.2 The book hoist

```
create the book hoist
  aka hoist, cage, lift
  a vehicle
  in the Undercroft
  movement: line
  carries: in
  drive: state
  stops: the Undercroft, the Library, the Lantern Gallery
  direction: both

  A wicker cage on a rope, in a shaft that goes up through the
  ↪ Library
  floor and on into the tower. It was made to take books to
  ↪ whoever
  wanted them, and it has never needed telling where that was.

  position
    when the book hoist holds the lens and the iron grille is
    ↪ lifted: the Lantern Gallery
    when the book hoist holds the lens: the Library
    otherwise: the Undercroft
  end position
```

`a vehicle` is a kind, like `a container`. The lines beneath it are its settings, each a word, a colon and a value.

`movement:` is required on every vehicle and says what sort of path it follows. A `line` is a fixed list of stops.

`carries:` is `in` or `on`, and decides whether the player gets into the vehicle or onto it. Things can be put there too, as into a container or onto a supporter.

`drive:` says what makes it move. There are three answers. `state` means the story's own conditions decide where the vehicle is. `control`

means statements start, stop and send it. `player` means the player steers.

`stops:` lists the rooms on the line, in order, and `direction:` is `both` or `one-way`.

33.2.1 Where a state-driven vehicle is

A `drive: state` vehicle has a `position` block. After every turn the platform reads its rows from the top, and the vehicle is at the stop named by the first row whose condition holds. The last row is always `otherwise`.

The hoist was made to take books to whoever wanted them. In this story it takes the lens to where the lens is wanted. Loaded, it rises to the Library. Loaded, with the ward on the stair lifted, it goes all the way to the roof. Empty, it comes home.

```
> put lens in hoist
You put the lens in the book hoist.

The book hoist pulls away.

> look
Undercroft
Low vaults on squat pillars, running off under the whole of the
hall. The air is cold and smells of stone. The steps go back up.

You can see a crate and a wisp here.
```

Ansel walks up, and the hoist is waiting.

```
> look
Lantern Gallery
The roof of the tower, open on every side. The Great Lantern
stands in the middle of it on a stone drum, taller than you are,
its glass thirty years dirty. To the east a low parapet is all
there is between you and the court. The stair goes back down.

You can see a book hoist and a wisp here.

In the book hoist you see a lens.

> put lens in lantern
(first taking the lens)

You take the lens from the book hoist.

You put the lens in the Great Lantern.

The lens drops into its seat with a sound like a coin on a
counter.

The book hoist pulls away.
```

The hoist is no way round the ward. Its second row sends it only as far as the Library while the grille stands, so the player who has not unbound the stair finds the lens one floor up and no nearer the roof.

A position block is the least work a vehicle can be. There is no control to write and nothing to steer. If the place a vehicle should be follows from the state of the story, say so, and let the platform move it.

33.2.2 Riding

The player can get into the hoist as well. These replies were recorded with the bell hushed. Ride while it is still tolling and each ends with the bell.

```
> enter hoist
You get into the book hoist.

> look
Undercroft
Low vaults on squat pillars, running off under the whole of the
hall. The air is cold and smells of stone. The steps go back up.

You can see a book hoist, a crate, and a wisp here.

In the book hoist you see Ansel.
```

Aboard a vehicle that is `carries: in` or `carries: on`, the player is still in the room the vehicle is in, and `look` describes that room. When the vehicle moves, the player moves with it.

33.3 A vehicle that is driven

`drive: control` puts the vehicle in the hands of statements. Here is a tram on a line of three stops.

```
create the tram
  a vehicle
  in the Yard
  movement: line
  carries: in
  drive: control
  stops: the Yard, the Shed, the Mill
  direction: both
  pace: 2 turns

  A tram.

create the lever
  a thing, scenery, pullable
  in the tram

  A lever.

  after the player pulling
    start the tram
  end after

create the brake
  a thing, scenery, pushable
  in the tram

  A brake.

  after the player pushing
    stop the tram
  end after
```

`start` and `stop` are the timer verbs of chapter 19, given a vehicle's name. Started, the tram runs along its line, taking the number of turns in `pace:` for each step, and stops by itself at the end.

```
> pull lever
You pull the lever.

> wait
Time passes...

The tram arrives at the Shed.
```

Two conditions read a vehicle. `the tram is under way` is true while it is moving, and `the tram is at the Shed` is true while it stands at that stop. Use them to gate a door, a description or a goal.

33.3.1 Flight

`movement: flight` is for a vehicle that goes from any of its places to any other without passing through the rooms between.

```
create the balloon
  a vehicle
  in the Yard
  movement: flight
  carries: in
  drive: control
  destinations: the Yard, the Mill
  pace: 2 turns

  A balloon.

create the cord
  a thing, scenery, pullable
  in the balloon

  A cord.

  after the player pulling
    send the balloon to the Mill when the balloon is at the Yard
    send the balloon to the Yard when the balloon is at the Mill
  end after
```

`destinations:` takes the place of `stops:`, and `send … to` names the one to make for. `pace:` here is the length of the whole trip. While the balloon is in the air it is nowhere, and a passenger who looks about sees the vehicle and its description.

```
> pull cord
You pull the cord.

> look
balloon
A balloon.

> wait
Time passes...

The balloon arrives at the Mill.
```

33.4 A vehicle the player steers

```
create the cart
  a vehicle
  in the Yard
  movement: compass
  carries: on
  drive: player
  commands: compass
  pace: 1 turn

  A cart.
```

A `compass` vehicle moves through the map by the rooms' own exits. With `drive: player` and `commands: compass`, a direction typed by a player who is aboard steers the vehicle and does not walk the player off it.

```
> enter cart
You get onto the cart.

> north
You steer the cart north.

The cart arrives at the Shed.
```

A steered vehicle keeps its heading. The cart goes on north, a step for each `pace:`, until the way ends or the player steers again, and then it "comes to a stop".

It will not enter a room that has not agreed to it. A room, a `rooms` group or a region says which steered vehicles it lets in.

```
create the Shed, a room
  admits: the cart
```

Without that line the cart stops at the Shed's door. This is deliberate. A story with a cart in it should not have to refuse the cart at the top of every staircase.

33.5 A vehicle with rooms inside

A large vehicle is a place. It has rooms, and the player walks about in them while it travels.

```
create the Cabin, a room
  up to the Deck

  A cabin.

create the Deck, a room
  west to outside

  A deck.

create the barge
  a vehicle
  in the Yard
  movement: line
  drive: control
  stops: the Yard, the Shed
  direction: both
  pace: 1 turn
  aboard: the Deck, the Cabin
  boarded through: the Deck

  A barge.
```

`aboard:` lists the rooms inside, and `boarded through:` names the one a player arrives in. A vehicle with rooms aboard has no `carries:` line. An exit written `to outside` leads from a room aboard to wherever the vehicle is at that moment.

```
> enter barge
You board the barge.

Deck
A deck.
```

After the barge has run to the Shed, `west` from the Deck leaves it there.

A vehicle that does not move at all is `movement: none`, with a `carries:` line and nothing else. It is a way of saying that something is a vehicle, for the sake of how it is entered and described, before deciding where it goes.

33.6 Build and try it

Copy `relighting.story` from the companion folder `chapter-33` over your own file. It is the story with this chapter's changes made, and it holds the one new phrase, which the chapter did not print. To make the changes by hand instead, work from this list and take the rest from that file: add the second blocked line to the Bell Stair, the book hoist to the Undercroft, and one phrase. The day's sequence is longer again, `at turn 100` and `60 turns later`, to leave room for the extra trip.

```
sharpee compose relighting.story --check
sharpee play .
```

Take the lens from the crate and try the stair with it: the grille answers until it is lifted, and the lens after. Then give the lens to the hoist, before and after the grille is lifted, and see where it goes.

> **UNDER THE HOOD: ONE ANSWER FOR EVERY MOVER · STORY IR `VEHICLE`**
>
> A vehicle's settings compile into a `vehicle` record on its entity, holding the movement type, the drive, and the stops, destinations or rooms aboard. The platform has a single rule for whether a mover may take a way from one room to another, and the walking player, a steered cart and a person on a goal all ask it the same question. That is why a blocked exit stops a cart as it stops Ansel, and why `admits:` is a line on the room and not on the vehicle. The line "The book hoist pulls away." and the arrival lines are standard messages, and can be overridden like any other.

FROM ANOTHER SYSTEM: VEHICLES

Inform 10 has a `vehicle` kind: an enterable container that the player, once inside, moves about the map by typing directions exactly as if on foot. That corresponds to Chord's steered `compass` vehicle. TADS 3 has a `Vehicle` class, a nested room that can move around with actors inside, and its documentation suggests building larger vehicles such as ships from one or more rooms. Chord's `line`, `flight` and `position` forms, in which a vehicle moves by itself, are not described in either manual.

33.7 What you have so far

The lens goes up the tower in a book hoist, which takes it as far as the wards allow. The stair will not pass it. A player who wants the Lantern lit must now lift the grille for two reasons.

Everything in the story is still a room, and the next chapter leaves rooms behind.

34 Open water and open ground

A map of rooms suits a building. It suits a lake badly. Open water has no doors and no corridors, and a story that cuts it into “Lake, North End” and “Lake, Middle” has made rooms where the player expected distance.

Chord has a second kind of place for this, the **expanse**: a stretch of open water or open ground measured in units, with rooms at fixed points on it and vehicles that cross it by heading. It is an opt-in system, so it needs a line in the story header. A second opt-in system, sailing, adds wind.

The Relighting uses both for its last scene. Lighting the Great Lantern has been the end of the game since chapter 18. From this chapter it is not quite the end.

34.1 Turning the system on

```
story
  title: The Relighting
  id: relighting
  use scoring
  use chapters
  use expanses
  use sailing
```

`use` lines were introduced with scoring in chapter 18, and the next chapter looks at them as a group. For now: `use expanses` admits the words of this chapter, and `use sailing` admits the ones about wind. Without the line, the compiler refuses the words and names the line that is missing.

34.2 An expanse

```
create the Mere
  aka water, lake
  an expanse
  size: 8 by 6
  wind: from west

  Grey water, wider than it looked from the roof, with a short
  ↪ chop on
  it.

  landmarks
    the Water Stair: 0, 2
    the Beacon Isle: 6, 2
  end landmarks
```

`an expanse` is a kind. `size:` gives its width and height in units, which are whatever you decide a unit is. Positions are counted from zero, east along the first number and north along the second.

The `landmarks` block places rooms on the expanse. A landmark is an ordinary room, with a description and things in it, that also has a position. The Mere has two: the Water Stair, at its western edge, and the Beacon Isle, six units to the east.

An expanse has no exits. Its four sides can be given blocked lines, as a room's exits can, and a vehicle that reaches a side with one is told so.

```
west is blocked: lake-edge
```

A side with no line simply ends: "You can go no farther that way."

34.2.1 Rooms seen from the water

A landmark needs one line that no other room has.

```
create the Water Stair, a room
  seen from afar: a dark arch at the foot of the Hall, with steps
  ↪ in it
  sight: 4

  Six steps going down into the Mere under an arch, and an iron
  ↪ ring in
  the wall for a boat. The white bar of the Lantern passes
  ↪ overhead and
  out across the water. The Undercroft is back to the south.

create the Beacon Isle, a room
  seen from afar: a hump of rock with a stone basket on a pole

  A rock the size of the refectory, with one thorn tree and the
  ↪ beacon.
  The Hall is a lit window on the far shore.
```

`seen from afar:` is how the room looks from a distance, and the compiler requires it of every landmark. `sight:` on a landmark is how far one can see out over the expanse from it.

The Water Stair is reached from the Undercroft by an ordinary exit, which is blocked until the Hall is awake.

```
create the Undercroft, a room, dark
  in the Hall
  north to the Water Stair
  north is blocked while the story is not awake: water-gate-shut
```

34.3 A vehicle on open water

```
create the skiff
  aka boat, heron
  a vehicle
  in the Mere at 0, 2
  movement: open
  carries: in
  drive: player
  commands: naval, compass
  stride: 1 unit
  sight: 5
  seen from afar: a small boat with a patched brown sail
  sails: furled, full
  sails start: furled

  A clinker skiff with one mast and one sail, both older than you
  ↪ are.
  Somebody has kept her tarred, and painted HERON on her transom.

  sailing
    0 to 44 degrees: furled stopped, full stopped
    45 to 119 degrees: furled stopped, full 2 turns
    120 to 180 degrees: furled stopped, full 1 turn
    calm: stopped
  end sailing
```

This is a vehicle of chapter 33 with a sixth movement type, `open`.

Its placement says where on the expanse it starts: `in the Mere at 0, 2`. That is the Water Stair's position, and a vehicle at a landmark's position is at that room. The player finds the skiff at the foot of the steps.

`stride:` is how many units one step covers. `sight:` is how far a person aboard can see, and `seen from afar:` is how the vehicle looks to someone else. An open vehicle must have all three.

Leave the sailing lines aside for a moment. An open vehicle without them takes a `pace:` line, as other vehicles do, and moves a stride for each pace in whatever direction it was last steered.

```
create the raft
  a vehicle
  in the Lake at 0, 1
  movement: open
  carries: on
  drive: player
  commands: compass
  stride: 2 units
  pace: 1 turn
  sight: 3
  seen from afar: a raft of four logs

  A raft.
```

34.4 Looking across water

On an expanse, `look` gives the expanse's own description and then everything in sight, each with a distance and a bearing. While the skiff lies at the foot of the steps it is in the Water Stair, and `look` there describes that room. This is the view one unit out, once the commands of the next section have put the skiff under sail.

```
> look
Mere
Grey water, wider than it looked from the roof, with a short chop
on it.

You can see a skiff here.

In the skiff you see Ansel.

The Water Stair, close by, to the west.

The Beacon Isle, on the horizon, to the east.
```

The platform sorts what can be seen into three bands by distance: `close by`, `in the distance`, and `on the horizon`. Anything beyond the looker's `sight:` is not listed. An expanse may carry a `visibility:` line, a number of units that caps everyone's sight, for fog or dusk.

The bands are conditions too.

```
phrase isle-near when the Beacon Isle is close by
phrase isle-far when the Beacon Isle is on the horizon
```

`is in sight` is true in any of the three bands. Each is judged from where the player is looking. For a test that does not depend on eyesight, an open vehicle's distance can be measured.

```
on every turn while the skiff is within 2 of the Beacon Isle, once
  phrase beacon-near
end on
```

`within <n> of` takes a landmark or a pair of coordinates.

34.5 Wind

`use sailing` adds a wind to an expanse and a table to a vehicle.

The Mere's `wind: from west` gives the direction the wind blows from, by compass point. `wind: calm` is no wind.

The skiff's `sailing` block replaces `pace:`. Each row is a range of angles between the vehicle's heading and the wind, where 0 is sailing straight into the wind and 180 is running straight before it. The row says how many turns a step takes at that angle, or `stopped`. The rows must cover 0 to 180 exactly once, and there is one more row for a calm.

`sails:` names the settings of the sails, from least canvas to most, and `sails start:` picks the first. With settings declared, each row of the table gives a pace for every setting. Read the skiff's table across: with the sail furled she goes nowhere at any angle; under full sail she will not go within 44 degrees of the wind, makes a step every two turns across it, and a step a turn with it behind her.

The wind is from the west and the beacon is due east. The crossing is the easy one.

```
> enter skiff
You get into the skiff.

> steer east
You steer the skiff east.

> wait
You wait.

> make sail
You make sail.
```

`commands: naval, compass` gives the player two sets of commands aboard. The compass set is chapter 33's: a bare direction steers. The naval set adds `steer <direction>` and the sail commands: `make sail`, `shorten sail` and `furl the sails`.

Steering alone moved nothing, because the sail was furled. Under sail the skiff runs east a unit a turn, and the isle is six units off. Every command is a turn, `look` among them. Type `look` now, straight after `make sail`, and it gives the view of the last section, with the stair close by. Two turns after it the view has changed, and the turn after that brings the skiff in. Without that first `look` the skiff is a unit behind what follows.

```
> look
Mere
Grey water, wider than it looked from the roof, with a short chop
on it.

You can see a skiff here.

In the skiff you see Ansel.

The Water Stair, close by, to the west.

The Beacon Isle, on the horizon, to the east.

> wait
You wait. The Hall has had thirty years of practice and is better
at it than you are.

> wait
You wait. Somewhere a shutter knocks, twice, and thinks better of
it.

> look
Mere
Grey water, wider than it looked from the roof, with a short chop
on it.

You can see a skiff here.

In the skiff you see Ansel.

The Beacon Isle, in the distance, to the east.

The Water Stair, on the horizon, to the west.

The rock comes up out of the dusk ahead, and on it a pole with a
basket at the top, black against the last of the sky.

> wait
You wait.

The skiff arrives at the Beacon Isle.
```

Arriving at a landmark’s position is arriving at the room.

34.6 Things adrift

A thing may be put on an expanse without a vehicle under it, if it `floats`.

```
create the bottle
  a thing, floats
  in the Lake at 3, 1

  A bottle.
```

An expanse with anything floating in it needs a `drift:` line, `drift: 1 unit every 2 turns`, and the platform then moves each floating thing at that rate and says so when the player can see it: “The bottle drifts.” A vehicle that reaches a floating thing “comes alongside” it and stops.

34.7 The new ending

The Lantern no longer wins the game. Its clause wakes the Hall and says what is still to do.

```
after the player sparking
  lose lens-shatters when not lantern-ready
  change the story to awake when lantern-ready
  award relit when lantern-ready
  phrase lantern-lit when lantern-ready
end after
```

Making the story `awake` does three things that were already written. It stops night from falling, since the day’s sequence loses only `when the story is not awake`. It opens the water gate. And it begins a fourth chapter, added to the `define chapters` block.

```
crossing - The Crossing
  begins when the story becomes awake
```

The win is on the far side of the water.

```
create the beacon
  aka basket, cresset
  a thing, scenery, light-source, starts unlit, wick
  in the Beacon Isle
  score answered worth 5

  An iron basket on a pole, packed with pitch-pine that has had
  ↪ thirty
  years to dry.

  after the player sparking
    award answered
    win beacon-lit
  end after
```

The beacon is a `wick`, the trait from chapter 24, so it answers the same word the lamp, the candles and the Lantern answer. The Lantern's score drops from 15 to 10 to pay for it, and the total is still 50.

```
> spark beacon
You say the word. A flame stands up on the wick, blue at first, as
if it had been waiting there with its coat on.

You say the word across six feet of air, and the pitch-pine takes
it. Behind you, over the water, the white bar of the Lantern
swings round and stops, pointing here. The two lights hold one
another. Somewhere between them the Mere is full of small blue
sparks, coming home.

You have risen to the rank of a Lamplighter.

[The story has ended. RESTART, RESTORE or UNDO]
```

The line about rank appears when this award takes the score to fifty. A player who left an award behind on the way, the refectory candles for instance, wins without it.

34.8 Build and try it

Copy `relighting.story` from the companion folder `chapter-34` over your own file. It is the story with this chapter's changes made, and it holds the phrases, which the chapter did not print in full. To make the changes by hand instead, work from this list and take the rest from that file. Add the two `use` lines and the fourth chapter. Create the Mere, its two landmark rooms, the skiff and the beacon. Give the Undercroft its north exit and blocked line, change the Lantern's `win` to a `phrase`, and adjust the two scores.

```
sharpee compose relighting.story --check
sharpee play .
```

Win the game as far as the roof. Then go down to the Undercroft, through the water gate, and across.

> **UNDER THE HOOD: A PLACE WITH COORDINATES · STORY IR USES**
>
> The two header lines appear in the compiled IR's `uses` list, and the compiler admits a system's words only when its name is there. On the platform an expanse is a single place, and everything on it has a position in units as well as a container. The rule from chapter 33 that decides whether a mover may take a way is the same rule here: an edge of an expanse is a way that is closed, and a landmark is a way that leads into a room. Sailing adds only a function from the angle off the wind to a pace.

FROM ANOTHER SYSTEM: PLACES THAT ARE NOT ROOMS

Neither manual describes a place measured in units. Both model the world as rooms joined by connections. For seeing one place from another, Inform 10 has regions and backdrops, and TADS 3 has a `Distant` class for an object too far away to interact with. Neither moves anything by heading and distance.

34.9 What you have so far

That is Part VI, and the game is complete. The Relighting begins in a porter's lodge and ends on a rock in a lake. Between them are four workings, three people and a wisp, a bargain, a machine with an interlock, a hoist that knows where it is wanted, and a boat. It can be won in one way and lost in three.

One chapter of this Part remains, and it adds nothing to the game. It is about the systems this story chose not to use.

35 The opt-in systems

The Relighting's header has four lines that begin with `use`.

```
story
  title: The Relighting
  id: relighting
  use scoring
    rank "Latecomer" at 0
    rank "Second-Year" at 10
    rank "Journeyman" at 30
    rank "Lamplighter" at 50
  use chapters
  use expanses
  use sailing
```

Each arrived in the chapter that needed it. This chapter is about the line itself, and about two systems the game does not use. Nothing here is added to The Relighting. A story about relighting a lantern does not need a fight in it, and it should not grow one to fill a chapter.

35.1 What `use` means

Most of Chord is always available. Rooms, things, people, clauses, timers, conversations and machines need no permission.

A few systems are larger than a word. They bring vocabulary of their own, they run something on every turn, and they print lines the author did not write. A story that does not want them should not pay for them or be surprised by them. Those systems are opt-in, and `use <name>` in the story header is how a story opts in.

A `use` line does two things. It admits the system's words, so that the compiler will accept `an expanse` or `combatant`. And it tells the platform to start the system when the story loads.

The rules are few. A `use` line goes in the story header and nowhere else. Each system is named once. All but two are a single line: `use scoring` carries its rank ladder beneath it, as chapter 18 showed, and `use hunger` carries its own lines, below. A system may need another, as sailing needs expanses, and the compiler says so.

Using a word without its line is an error that names the line. So is a `use` line for something that needs none. People and machines are core, and `use npc` is refused.

Line	What it brings	Chapter
`use scoring`	`score`, `award`, the rank ladder	18
`use chapters`	`define chapters`, `during`, `before`, `after`	19
`use expanses`	`an expanse`, landmarks, `movement: open`	34
`use sailing`	`wind:`, the `sailing` block, `sails:`	34
`use combat`	`combatant`, `weapon`, fighting	this one
`use hunger`	a hunger meter with rungs	this one

35.2 Combat

`use combat` brings two trait words. A person who can fight is a `combatant`, and a thing to fight with is a `weapon`.

```
story
  title: The Cellar
  id: cellar
  use combat
  authors:
    Nobody

create the Cellar, a room

  A low brick cellar that smells of coal and damp.

create the rat
  a person, combatant with health 3 and skill 20 and base-damage 1
  in the Cellar

  A rat the size of a boot.

create the poker
  a thing, weapon with damage 3
  in the Cellar

  An iron poker.

create Ansel
  a person
  playable
  starts in the Cellar

  You came down for coal.

before the game starts
  change the player to Ansel
end before
```

That is a whole story, with a room, a player and a start block as Part I taught them. Both words take settings, written with `with` and `and` as the behaviour words of chapter 26 do. `health` is how much damage a combatant can take. `skill` is how likely its blows are to land, and `base-damage` is what a blow does bare-handed. A weapon's `damage` is what it adds.

The standard `attack` action has been in the grammar all along. Against something that is not a combatant it does nothing of consequence. Against a combatant it is a fight.

```
> attack rat
You land a solid blow on the rat, dealing 1 damage!

> take poker
Taken.

> attack rat with poker
You have slain the rat!
```

An attack may also miss: “You swing at the rat but miss!” Whether it does is a roll against skill. `sharpee play` rolls afresh each time it starts, so your fight will not go blow for blow as this one did, and the rat may take more attacks or fewer. A test tree is different. It carries a seed, as chapter 38 describes, and the rolls come from that seed, so a tree that pins a fight sees the same fight every time.

All of these lines are standard messages. A story with combat in it will want its own, and chapter 22’s `override message` is how.

The appendix of trait words lists every setting the two words take. Reach for combat when fighting is part of what your game is. A single guard who must be got past is better written as a blocked exit and a clause, with prose you control, than as a system of numbers that the player meets once.

35.3 Hunger

`use hunger` gives the player a meter that rises with time.

```
story
  title: The Yard
  id: yard
  use hunger
    grows 1 each turn
    peckish at 2 says hunger-peckish
    hungry at 4 says hunger-hungry
    fatal at 9
  authors:
    Nobody

create the Yard, a room

  A cobbled yard with a pump in the middle of it.

create Ansel
  a person
  playable
  starts in the Yard

  You have not eaten since yesterday.

before the game starts
  change the player to Ansel
end before

define phrases en-US
  hunger-peckish:
    You could eat.
  hunger-hungry:
    You are hungry now, and thinking about the loaf.
```

The lines beneath `use hunger` are its settings. `grows <n> each turn` is the rate. Each line of the form `<name> at <n>` is a **rung**: a level of the meter with a name of your choosing. A rung may end with `says` and a phrase, and the phrase is printed once, when the meter first reaches the rung. `fatal at <n>` is the level at which the player dies.

The meter starts at nothing. The opening of a story is not a turn, so the meter still stands at nothing when the player first reads the Yard, and each command after that adds one. The second `wait` takes it to two.

```
> wait
Time passes...

> wait
Time passes...

You could eat.

> wait
Time passes...

> wait
Time passes...

You are hungry now, and thinking about the loaf.
```

Left alone, the meter reaches nine on the ninth `wait`.

```
> wait
Time passes...

You have starved to death.

[The story has ended. RESTART, RESTORE or UNDO]
```

This is the form of chapter 18's rank ladder, turned upside down. A rank is a threshold on a number that rises as the player does well, and a rung is a threshold on a number that rises as time passes. Both speak once on the way up.

35.3.1 Announcing

A system that crosses thresholds has to decide how much to say about it. A `use` line may end with `, announce` and a mode.

```
use hunger, announce silent
```

`silent` prints nothing at a rung. The meter still rises, and the fatal level still ends the game with its line. The other modes are `all`, which is the default and reports every rung as it is reached, and `collapsed` and `combined`, which say less when several rungs are crossed at once.

The suffix is allowed on any `use` line, and it matters for the ones that count something, which are hunger and scoring.

35.4 Choosing

An opt-in system saves work when its model is the model your game has. Scoring is that for most games, and The Relighting used it without a second thought. Expanses were that for one scene of this game and would have been absurd for the rest.

When a system's model is nearly what you want, be careful. You will spend the saved work bending it. A meter is a counter, a rung is a comparison, and a starving player is `kill the player` with a `when` on it. Chapter 18 gave you all three, and a story that wants a thirst that only matters in one desert can write it in ten lines and own every word.

35.5 Build and try it

There is no companion snapshot for this chapter. The two examples above are complete stories, and you can build them as you built the first story in Part I. Each needs a folder of its own, because a folder holds one story. Make a folder called `cellar` beside your project, and later one called `yard` in the same place. Type the first example into a file called `cellar.story` inside it, and run these two commands there.

```
sharpee compose cellar.story --check
sharpee play .
```

Attack the rat, take the poker, and attack again until the rat is dead. Then make a folder called `yard`, type the second example into `yard.story` inside it, and do the same.

```
sharpee compose yard.story --check
sharpee play .
```

Type `wait` until the story ends, and count the turns between the lines.

UNDER THE HOOD: A MANIFEST FOR EACH SYSTEM · STORY IR `USES`, `ANNOUNCEMODES`

Each opt-in system is described to the compiler by a small manifest: its name, the trait words it adds, and the settings each word takes with their types. That is how `weapon with damage three` can be an error at build time though the compiler knows nothing about fighting. The compiled IR lists the systems a story opted into under `uses`, and any `announce` suffix under `announceModes`. When the story loads, the platform starts one plugin for each name in `uses`, and each plugin takes its turn in the cycle after the player's command.

FROM ANOTHER SYSTEM: EXTENSIONS

Both Inform 10 and TADS 3 handle this with libraries that a story includes. An Inform story writes "Include", the name of an extension and its author, as in "Include Rideable Vehicles by Graham Nelson." A TADS 3 project names its libraries in the project file, with a `-lib` line for each. A Chord `use` line is narrower than either. It turns on a system that ships with the platform, and it cannot name code that the story's author wrote.

35.6 What you have so far

That is Part VI. You have every part of the language that builds a world and makes it move: rooms and things, clauses and conditions, phrases and verbs, people who act and talk and feel, machines, vehicles, open water, and the systems a story may opt into.

Two things remain. The player meets a story through a page, with sound and pictures and a layout, and Part VII is about that page. And a story that you mean other people to play has to be proved and shipped, which is Part VIII.

PART VII. PRESENTATION

36 Sound, music and pictures

Everything the story has said so far, it has said in text. A terminal can do nothing else, and a great deal of interactive fiction wants nothing else. A browser can do more. It can play a recording, loop a bed of sound under a scene, and put a picture on the page.

Chord treats all of that as something the story offers and the player's program may decline. A story declares its sounds and pictures, and says when to play and show them. A client that can do it does it. A client that cannot leaves it out, and the story plays on, because nothing in the world ever depends on whether a file was heard.

This chapter gives The Relighting a recording of the bell, the sound of a flame catching, water under the skiff, and one picture.

36.1 Declaring an asset

An **asset** is a name the story uses and a file the client fetches.

```
define sound tolling from "audio/toll.mp3"

define sound lapping from "audio/lapping.mp3"

define image lantern-lit-picture from "img/lantern-lit.png"
```

`define sound`, `define image` and `define music` are top-level lines. Each gives one word as the name and one quoted path. The path is relative to a folder called `assets` beside the story file, so the first line above means `assets/audio/toll.mp3`.

The compiler checks the names and not the files. A statement that plays a sound nobody declared is an error, and so is one that shows a sound as if it were a picture. A path to a file that does not exist is not noticed until

the story is built for the browser, in chapter 40. You can write this chapter's lines, and play the story in a terminal, before any of the recordings exist.

36.2 Playing a sound

```
on every turn while the tower bell is tolling
  select cycling
    phrase toll-near
  or
    phrase toll-far
  or
    phrase toll-count
  end select
  raise tolls by 1
  play sound tolling
end on
```

`play sound` is a statement, and it goes wherever a statement goes. This one is the last line of the header's `on every turn` block from chapter 19, so the recording is played on every turn the bell tolls.

In a terminal you will see no difference at all. The story's own three sentences about the bell still print, and the recording is offered to a client that is not there. That is the right result, and it is why the sentences stay. A sound should never be the only way a player learns something. Some players have the sound off, some cannot hear it, and some are reading your story through a screen reader that is already talking.

36.3 A sound is also a sound in the world

Chapter 10 said that sound goes further in Chord than a `loud` thing, and this is where. A sound that a story plays is two things at once. It is a cue for the client, and it is a noise that happens at a place in the story's world, where people can hear it.

The place is wherever the statement's owner is. A `play sound` in a thing's clause sounds in that thing's room. The wisp can be made to notice.

```
create the wisp
  aka light
  a person, follower, listener
  pronouns it

  A scrap of light the size of a thumbnail, the same blue as the
  ↪ lamp
  flame. It keeps a little behind your shoulder, where a dog
  ↪ would.

  after hearing the tolling, once
    phrase wisp-flinches
  end after
```

`listener` is a trait word for a person. A listener is told about each sound that reaches the room it is in. `after hearing the <sound>` is a clause that runs when one does, and it takes `once` as other clauses do.

```
> light lamp with taper
You have no match, and you did not walk here for three days to use
one. There is a word for this.

In the lodge, an old man who was not asleep opens his eyes.

You are Hobb, and that is the bell. It has not rung since the
night you stopped it. There was a word for stopping it, and you
wrote the word down, because you knew you would not keep it in
your head, and you put the paper somewhere safe.

The tower bell tolls.

Somewhere in the Hall a small blue light goes flat and dim at the
first stroke, like a flame in a draught, and creeps back to its
size.
```

Read the last sentence again with chapter 31 in mind. On the turn the bell first tolls, the story hands the player's role to Hobb, who is in the Lodge. The wisp is in the Library. A clause that runs `after hearing` prints its prose whether or not the player can see the listener, so the sentence had to be written to be true from anywhere. Write a listener's reaction as something that could be overheard, or keep it to a change of state and let a description report it later.

36.4 What a sound sounds like

The player hears sounds too. What the player reads is up to the sound's declaration, which may be a block.

```
define sound catch from "audio/catch.mp3"
  heard: The wick takes with a small dry sound, like a page
  ↪ turned.
end sound
```

A sound block closes with `end sound` and may hold a `heard:` line, a `heard from afar:` line, or both. The `heard:` line is what the player reads when the sound is made nearby. This sound belongs to the `wick` trait of chapter 24, so every lamp, candle and lantern in the story now catches audibly.

```
define trait wick
  on the player sparking
    refuse when it is lit: already-burning
    change it to lit
    phrase sparked
    play sound catch
  end on
```

```
> light lamp
You say the word. A flame stands up on the wick, blue at first, as
if it had been waiting there with its coat on.

Something small and blue comes loose from the flame and hangs in
the air beside it, undecided. Then it decides.

Somewhere above you, old timber takes a weight it had forgotten.

The wick takes with a small dry sound, like a page turned.
```

A sound prints only what its declaration states. The tolling has no block, so it prints nothing, and the story's own bell sentences are not doubled. Give a sound a `heard:` line when the sound is the news. Leave it off when some other prose already carries the news and the recording is decoration.

36.5 From afar, and with no file

A sound does not need a recording. Leave out `from` and the sound exists in the world only: listeners hear it, the player reads its lines, and no client is asked to play anything.

```
define sound prising
  heard: Iron shrieks on iron, close by, and stops.
  heard from afar: Somewhere in the Hall, iron complains about
  ↪ iron.
end sound
```

When the bell begins, Tamsin Vey goes to the grille on the Bell Stair and tries her pry bar on it. Chapter 26 gave her that goal. The grille can answer for it.

```
create the iron grille
  aka grille
  a thing, scenery, ward
  in the Bell Stair
  states: standing, lifted

  after Tamsin Vey pushing
    play sound prising
  end after
```

The clause is on the grille, so the sound is made in the Bell Stair. What the player reads depends on how far away the player is. In the same room, or one open doorway from it, the sound arrives whole and the `heard:` line prints. Further off, or through a closed door, it arrives weakened, and the `heard from afar:` line prints. Far enough away, nothing prints.

The block below is what the player reads on walking from the Cloister into the Bell Stair on the turn she pushes. That one turn is easy to miss. Wait in the Cloister instead and the last line prints by itself, after a `wait`.

```
> north
Bell Stair
The foot of the tower. A stair winds up around the well where the
bell rope hangs. The cloister is south. An iron grille stands
across its third step.

You can see Tamsin Vey here.

The bell tolls again, and the rooks go up off the roof.

The wisp follows you.

Tamsin Vey pushes the iron grille.

Iron shrieks on iron, close by, and stops.
```

Waiting in the Library, two rooms from it:

```
> wait
You wait.

The bell tolls again, and the rooks go up off the roof.

Somewhere in the Hall, iron complains about iron.
```

A sound that states only one of the two lines prints nothing at the other distance. That is often what you want. A match being struck has a `heard:` line and no business being heard from the next county.

36.6 A bed of sound

A sound effect plays once and is over. An **ambient** bed loops until the story stops it.

```
create the skiff
  aka boat, heron
  a vehicle
  in the Mere at 0, 2

  after the player entering
    play ambient lapping
  end after

  after the player exiting
    stop ambient
  end after
```

`play ambient` takes the name of a sound asset, which must have a file. `stop ambient` stops it. Nothing prints for either, and nobody in the world hears a bed: it is for the player's ears only, and it says nothing that the description of the Mere has not said.

36.7 Showing a picture

```
after the player sparking
  lose lens-shatters when not lantern-ready
  change the story to awake when lantern-ready
  award relit when lantern-ready
  phrase lantern-lit when lantern-ready
  show image lantern-lit-picture when lantern-ready
end after
```

`show image` puts a declared image on the page, and `hide image` takes it away. Like every statement, each may end in a `when` condition. The picture of the lit Lantern stays up for the crossing of the Mere, and the beacon's clause hides it on the last turn.

```
after the player sparking
  award answered
  hide image
  win beacon-lit
end after
```

Where the picture appears on the page, and how large, is the page's business. That is the next chapter.

36.8 Music, named beds and layers

The Relighting has no music. A story that does declares it the same way and has two statements for it.

```
define music waltz from "audio/waltz.mp3"
define sound rain from "audio/rain.mp3"
define sound hiss from "audio/hiss.mp3"
define image portrait from "img/portrait.png"
define image fog from "img/fog.png"

define ambient weather
define layer window

create the gramophone
  a thing, scenery, switchable
  in the Parlour

  A gramophone with a horn like a brass lily.

  after the player switching_on
    play music waltz looping
    play ambient hiss
    phrase needle-down when not client has music
  end after

  after the player switching_off
    stop music
    stop ambient
  end after
```

`play music` starts a track, and `looping` makes it repeat. `stop music` ends it. There is one music track at a time.

There may be several beds. `play ambient hiss` uses the bed every story has, which is called `main`. `define ambient weather` declares a second bed by name, and `play ambient rain in weather` and `stop ambient in weather` address it. The two loop together and stop separately.

Pictures have **layers** in the same way. Three exist without being declared: `background`, `main` and `overlay`. `show image` with no layer uses `main`. `define layer window` adds one, and `show image fog in window` shows a picture in it.

```
create the curtains
  a thing, scenery, openable
  in the Parlour

  Heavy curtains.

  after the player opening
    play ambient rain in weather
    show image fog in window
    show image portrait
  end after

  after the player closing
    stop ambient in weather
    transition fade
    clear
  end after
```

Two more statements are for the page as a whole. `transition` followed by a word asks the client for a visual transition of that kind, and `clear` asks it to clear the screen.

36.9 Asking what the client can do

The gramophone's clause has a condition in it that reads the client and not the world.

```
phrase needle-down when not client has music
```

`client has` is followed by one word from a closed list of capabilities, among them `sound`, `music`, `images`, `transitions`, `layers`, `split-pane`, `status-bar` and `sidebar`. It is true when the program the player is using has said it can do that thing. In a terminal the gramophone prints its sentence, because a terminal has no music.

```
> switch on gramophone
The gramophone hums to life.

The needle comes down, and a waltz gets up out of the crackle.
```

This is the pattern for anything a recording was going to carry alone: play the recording, and say the same thing in words to a player who has no recording.

One statement may not be made to depend on the client. `play sound hiss when client has sound` is an error, `analysis.play-sound-client-has`. A played sound happens in the world, where listeners react to it, and the world must not go differently for two players because one of them has speakers. Leave the condition off. A client without sound does not play the cue, and that is all.

36.10 Build and try it

Copy `relighting.story` and the `assets` folder from the companion folder `chapter-36` into your project, the story over your own file and the folder beside it. The story is yours with this chapter's changes made, and the folder holds three short recordings and a picture. To make the changes by hand instead, work from this list and take the rest from that file: add the five declarations at the top of the story, `play sound catch` to the `wick` trait, `play sound tolling` to the header's block, the clause to the grille, `listener` and its clause to the wisp, the two clauses to the skiff, and the `show image` and `hide image` lines.

```
sharpee compose relighting.story --check
sharpee play .
```

Light the lamp and read its last line. Then wait in the Library for the bell, look in the newspapers as Hobb, and keep waiting as Ansel until iron complains about iron. Nothing will play in the terminal. Chapter 40 builds the page where it does.

UNDER THE HOOD: MEDIA EVENTS · EVENT IDS

`MEDIA.SOUND.PLAY`, `MEDIA.AMBIENT.PLAY`, `MEDIA.IMAGE.SHOW`

Each statement in this chapter becomes one event with a small payload. `play sound tolling` sends `media.sound.play` with the file's path as `src`. `play ambient lapping` sends `media.ambient.play` with the path and the bed's name, and `show image` sends `media.image.show` with the path and, when one is named, the layer. The stops and the hide are `media.ambient.stop`, `media.music.stop` and `media.image.hide`. A played sound also raises the world's own sound event, which carries the thing that made it, the room, and the sound's name, and it is that event a listener's `after hearing` clause answers. A test tree can assert on any of these by name, as chapter 38 shows.

FROM ANOTHER SYSTEM: FIGURES AND SOUNDS

Inform 10 declares a picture with a sentence such as `Figure of Woodlands is the file "Woodlands.png".` and shows it with `display the Figure of Woodlands;`. A sound is `Sound of rustling leaves is the file "Rustling leaves.ogg".` and `play the sound of rustling leaves;`. *Writing with Inform* says both can be used only when the project is compiled to the Glulx story file format. TADS 3 does it in the output text, with HTML TADS tags: `<IMG SRC="picture.jpg">` for a picture and a `<SOUND>` tag for a sound, and a game asks `systemInfo()` what the interpreter can do. In neither system is a played sound an event in the story's world that a character can hear, on the pages read for this comparison.

36.11 What you have so far

The Relighting can be heard. The bell has a recording and the wisp flinches at its first stroke. A flame catching makes a sound the player reads. A pry bar on the grille is heard plainly from the Cloister and faintly from the Library. The skiff has water under it, and the lit Lantern has a picture.

None of it has been heard or seen yet, because a terminal cannot. The picture needs a place on a page, and the page needs to be told which of its features this story uses. The next chapter is about the page.

37 The page the player sees

A story played in a terminal has no say in how it looks. A story played in a browser has a page, and the page has a title bar, a status line, a place for a picture, a column for panels, a menu of themes, and somewhere to keep saved games.

Chord gives a story three ways to shape that page. The header chooses the client and its themes. A **channel** sends a named value to the page, beside the prose, for the page to show where it likes. And a folder beside the story holds a stylesheet, and if you want one, a page of your own.

This chapter builds The Relighting for the browser for the first time and gives it a panel that lists the workings Ansel has learned.

37.1 The client fields

The story header from chapter 2 has a second group of fields, which say nothing about the story and everything about where it will be played.

```
story
  title: The Relighting
  id: relighting
  story-version: 0.7.0
  client: browser
  themes: paper, modern-dark
  default-theme: paper
  storage-prefix: relighting
```

`client` names the program the story is built for. There is one value today, `browser`, and it is also what a story gets by leaving the line out. Any other word is refused when the story is built.

```
Error: unknown client 'terminal' (story header `client:`) — the
build can only produce 'browser'.
```

`themes` is a list of themes, separated by commas, that the page will offer in its Settings menu. Four are built in: `paper`, `modern-dark`, `retro-terminal` and `system-6`. The page always offers a fifth, called Classic, which is the page with no theme applied.

`default-theme` is the theme the page opens in. It should be one of the names in `themes`. Leave it out and the page opens in Classic.

`storage-prefix` is the word the browser files this story's saved games under. It defaults to the story's `id`, so most stories never write it. It matters when two stories are served from one site, because the browser keeps one drawer per site and the prefix is what keeps their saves apart.

The compiler accepts any word as a theme name. The build is where a wrong one is caught.

```
Error: Unknown built-in theme "parchment". Available: modern-dark,
retro-terminal, paper, system-6. For your own theme, list { "id":
"…", "name": "…" } and define its [data-theme] block in browser/
<package-name>.css.
```

Only the `themes` list is checked in this way. A `default-theme` that names a theme the list does not carry builds without complaint and opens on a page with no stylesheet for it, so read the two lines together.

37.2 Building the page

```
sharpee build relighting.story
```

The command compiles the story, as `compose` does, and then writes a complete web page to a folder under `dist/web/`, named for the story's `id`. There is no need to run it yet. "Build and try it" runs it at the end of

the chapter, on the story with everything this chapter adds, and this is what it prints then.

```
🔨
   Building browser bundle

  ✓ Minted the story's ifid into relighting.config.json
  Story: The Relighting (relighting)
  ✓ Validated relighting.story (gate-clean) — source not shipped
(`publish-source:` absent)
  ✓ Story IR → dist/relighting.ir.json
  Generated browser entry from template
  ✓ Embedded the compiled story IR into the bundle
  Bundling game.js...

  dist/web/relighting/game.js      1.9mb
  dist/web/relighting/game.js.map  3.5mb

⚡
   Done in 87ms
  ✓ Built game.js
  ✓ Copied index.html
  ✓ Copied index-testing.html (IDE testing surface — excluded from
publish)
  ✓ Copied platform engine CSS (base, engine, decorations)
  ✓ Wired 2 theme(s): paper, modern-dark
  ✓ Copied relighting.css
  ✓ Copied assets/ (2 entries)

✅
   Build complete! (game.js 1907.7 KB)
Output: dist/web/relighting/

To test locally:
  npx serve dist/web/relighting
```

The line about `relighting.css` appears once the stylesheet later in this chapter exists, and the line about themes once the header names them. The line about an `ifid` is chapter 40's subject. It is printed by the first build of a story that has no identifier yet.

The folder holds `index.html`, one script called `game.js` with the whole story inside it, the stylesheets, and the `audio` and `img` folders that were inside the `assets` folder of chapter 36. Nothing in it needs a server that

does more than hand out files. To look at it on your own machine, serve the folder and open the address it prints.

```
npx serve dist/web/relighting
```

The page opens on the story's title and prologue, in the paper theme. The Settings menu has a Theme entry that lists Classic, Paper and Modern Dark. The page has no sound control of its own, so see that your machine's sound is up before you spark the lamp. The first build also writes a small file beside the story, `relighting.config.json`, which chapter 40 explains.

The page also prints something the terminal does not. When one of the chapters from chapter 19 begins, the page prints its title in bold, and its description in italics beneath if it has one, above the reply to the command that began it. `Morning` stands over the first room, and reading the spark page begins the afternoon.

```
> read spark page

Afternoon

The Hall has noticed you.

(first taking the spark page)
```

37.3 A channel

The prose is not the only thing a story sends to the page. The room's name in the status line, the score and the turn count each travel on a **channel**: a named value that the page shows in a place of its own. Those channels are the platform's. A story may define more.

```
define channel workings
  mode append
  return "(working): (gloss)" from relighting-working-learned
end channel
```

`define channel` is a top-level block with a one-word name. It has two lines that matter.

The `return` line says what the channel's value is and when it changes. It ends in `from` and the name of an **event**. Each time the story raises that event, the channel takes a new value, built from what the event carried. Here the value is a piece of text with two slots in parentheses, and each slot is filled from the event's field of that name.

The `mode` line says what a new value does to the old one.

- `mode replace` keeps only the latest value. A tide gauge, a compass heading, or the name of a chapter is a replace channel.
- `mode append` adds each value to a growing list. A log, a list of clues, or this list of workings is an append channel.
- `mode event` has no lasting value at all. It reports that something happened this turn, and the page reacts and forgets.

37.4 Raising an event

A story raises an event with `emit` .

```
on the player reading
  select on the spark page's state
    when legible
      change the spark page to learned
      change the story to waking
      start waking
      raise workings by 1
      emit relighting-working-learned with working "spark" and
      ↪ gloss "wakes a flame"
      award first-working
      phrase spark-learned
```

`emit` is a statement. It names the event with one hyphenated word, then `with`, then the event's fields joined by `and`. A field is a name followed by its value: a quoted string, a number, or a reference to something in the world.

The event's name is the story's own, and the convention is to begin it with the story's `id`, so that it cannot collide with an event of the platform's. The name is one word. A dotted name is an error, `parse.dotted-key`.

Each of the four pages emits the same event with its own two strings. The channel hears all four.

An event is not prose, and `emit` prints nothing. In a terminal, reading the spark page looks exactly as it did in chapter 13. On the page, a column opens beside the story.

```
workings
spark: wakes a flame
```

By the time the Lantern is lit the panel holds four lines, in the order Ansel read the pages.

```
workings
spark: wakes a flame
unbind: releases a ward
hush: quiets a thing
mend: makes a broken thing whole
```

A channel the page has no special place for is shown in this way, as a titled panel in a column at the side. The column is hidden until some channel has a value, so a player who never reads a page never sees an empty box.

37.5 What a channel may return

The `return` line has four forms. The first three give the channel a single value.

```
define channel tide
  mode replace
  return level from harbour-tide-turned
end channel

define channel bell
  mode event
  gated by sound
  return phrase bell-caption from harbour-bell-rung
end channel
```

`return level from …` takes one field of the event, by name, and the channel's value is whatever that field held: `high` after `emit harbour-tide-turned with level "high"`, and the number 3 after `emit harbour-tide-turned with level 3`.

`return "…" from …` is the text form the workings channel uses.

`return phrase bell-caption from …` gives the channel a phrase from the story's phrase table, so the value is prose that can vary as any phrase can.

`gated by` names one of the client capabilities of chapter 36. A gated channel is sent only to a client that has the capability. The bell's caption is for a page that is about to play the bell.

The fourth form returns a **record**: several named values at once.

```
define channel card
  mode replace
  return record from harbour-card-dealt
    suit suit
    rank rank
    notes list of note
  end record
end channel
```

Each line of the record is a member: the member's name, then where its value comes from, which is again a field name, a text with slots, or `phrase` and a key. `list of` makes the member a list, even when the event carried a single value. After `emit harbour-card-dealt with suit "spades" and rank 7 and note "dog-eared"`, the channel's value is a record whose `suit` is `spades`, whose `rank` is 7, and whose `notes` is a list of one string.

A record is for a page of your own, which can lay the members out as it likes. A panel a player reads is better served by a text.

37.6 A reference in a field

An `emit` field may hold a reference to a thing or a room, with no quotation marks.

```
emit harbour-noted with what "gauge read" and where the Quay
```

The event does not carry the Quay's name. It carries the Quay's identifier, which for this room is `quay`, and a channel that returns `"(what): (where)"` shows `gauge read: quay`. That is useful to a page

that keeps its own table of rooms, and wrong for a panel a player reads. When the value is for reading, send a string.

37.7 The stylesheet

A story's own styles go in a folder called `browser` beside the story file, in a stylesheet named for the story's `id`.

```
/* browser/relighting.css */

#channel-panel-workings .sharpee-channel-panel-title {
  font-variant: small-caps;
  letter-spacing: 0.06em;
}

#channel-panel-workings .sharpee-channel-panel-row {
  margin-bottom: 0.4rem;
}
```

The build copies it into the page's folder and loads it last, after the platform's styles and after the themes, so a rule here wins. A channel's panel has the identifier `channel-panel-` followed by the channel's name.

Keep colours out of this file where you can. The themes set them, and a colour fixed here will be right in one theme and wrong in the other.

The picture from chapter 36 appears above the story, in a strip whose height the platform's stylesheet sets. A rule for `#sharpee-media` in your own stylesheet changes it. A chapter's title and description are printed in a block with the class `sharpee-chapter`, and a rule for that class restyles the block or hides it.

37.8 A page of your own

The stylesheet is enough for most stories. For the rest, the whole page can be replaced. The default `index.html` in the build folder opens with a

comment that says how: copy it to `browser/index.html` beside your story and edit it there. From then on the build uses your page as it stands.

Two elements must keep their identifiers, because the client looks for them by name: `sharpee-media`, where pictures go, and `sharpee-sidebar`, where panels go. You may move them anywhere on the page. The classes that begin `sharpee-` are what the themes style, so keep those on anything you want themed.

Start with the stylesheet. A copied page is yours to maintain, and it will not pick up improvements to the default one.

37.9 Build and try it

Copy `relighting.story`, `relighting.config.json` and the `browser` folder from the companion folder `chapter-37` into your project, the story over your own file and the other two beside it. The story is yours with this chapter's changes made, and its `story-version` is `0.7.0`. The copied story already carries its identifier, in the header and in the config file, so your build does not print the line about minting one that the build earlier in this chapter shows. To make the changes by hand instead, work from this list and take the rest from that file: add the four client fields to the header, the `define channel` block near the top of the story, and one `emit` line to each of the four pages' `on the player reading` clauses, and make the folder `browser` and put `relighting.css` in it.

```
sharpee compose relighting.story --check
sharpee build relighting.story
npx serve dist/web/relighting
```

Play as far as the Library and read the spark page. Then change the theme from the Settings menu and reload the page. The theme you chose has been remembered. It was filed in the browser under the storage prefix, as `relighting-theme`, and the saved games are filed beside it.

The reload also brings the game back from its autosave, with the line `[Session restored]`. The status line names the Library again, and the `workings` panel is there with its one line, because the page saves what its channels were showing along with the game.

UNDER THE HOOD: CHANNELS · STORY IR `CHANNELS`, PLATFORM CHANNELS `LOCATION`, `SCORE`, `TURN`

The compiled IR has a top-level `channels` list holding each channel's name, mode, gate and return. At the end of every turn the platform asks each channel, its own and the story's, whether this turn's events give it a value, and sends the page the values that changed. The status line is three platform channels, `location`, `score` and `turn`. The pictures of chapter 36 are channels as well, one for each layer, and the sounds another. A story's `emit` adds an event to the turn like any other, with its fields as the event's data.

FROM ANOTHER SYSTEM: THE STATUS LINE AND OTHER WINDOWS

Inform 10 gives a story the two ends of the status line as values it can set: `now the left hand status line is "[the player's surroundings] / [turn count] / [score]";`. TADS 3 has the banner window, which its System Manual describes as "a rectangular area of the screen that can be separately controlled: each banner has its own contents, background color, text color, and so on", and a game creates one with `bannerCreate()` and writes to it. In both, the story writes the text that appears in the window. A Chord channel sends a value and leaves its appearance to the page.

37.10 What you have so far

The Relighting has a page. It opens in a theme the story chose, offers a second, keeps its saves under its own name, plays three recordings, shows a picture when the Lantern is lit, and lists the workings in a panel that grows as Ansel learns them.

That is Part VII, and the story is finished. Nothing more will be added to it. What is left is to prove that it works, and will go on working after the next change, and then to give it to somebody. Part VIII begins with the proof.

PART VIII. PROVING AND SHIPPING

38 Test trees

The Relighting is finished, and it has been played to its end many times in the writing of this book. Each of those plays proved something on the day. None of them proves anything about tomorrow, when you move a key or reword a phrase and something four rooms away stops working.

A **test tree** is a play-through that has been written down, with a note on each turn of what the story must say. The `sharpee test` command plays it again, turn by turn, and tells you which notes no longer hold. It is called a tree because a play-through can fork: from one position in the story you may record several different continuations, and they share everything before the fork.

This chapter builds a small tree by hand and then reads the full one that comes with the story.

38.1 Two ways to make one

Chord Writer has a Testing tab that records. You play the story there, and each turn becomes a card that you can pin an expectation to.

The `sharpee` command has no recorder. From the command line a tree is written as files, by hand, and the story's own output is read back from a test run. That sounds laborious and is less so than it sounds, because a tree is very regular. It is also worth learning whichever tool you use, because the files are the same, they live beside your story, and they are what you will read when a test fails.

38.2 The folder

A story's tree is a folder beside the story file, named for the story's `id` with `.tests` on the end. For The Relighting it is `relighting.tests`. It

holds one file called `manifest.json` and one file for each stretch of the play-through.

```
{
  "seed": 42,
  "story": "relighting",
  "version": 3
}
```

That is the whole manifest. `story` is the story's `id`. `version` is the version of the tree format, which is 3. `seed` fixes the story's luck: every `one chance in` condition and every `select random` of chapter 17 falls the same way on every run at the same seed, so a tree that passes once passes always, until the story changes.

38.3 A segment and its cards

The play-through itself is in **segment** files. A segment is a run of turns with no fork in it. Its file is named for its identifier, and the identifier is exactly eight lowercase letters or digits.

Here is `mainline.json`, a first segment of three cards.

```
{
  "cards": [
    {
      "id": "card0001",
      "skip": true,
      "type": "opening"
    },
    {
      "command": "take lamp",
      "id": "card0002",
      "skip": true,
      "type": "turn"
    },
    {
      "command": "open press",
      "id": "card0003",
      "skip": true,
      "type": "turn"
    }
  ],
  "id": "mainline"
}
```

A **card** is one step. There are two types. The `opening` card is everything the story prints before anything is typed: its prologue, its title and credits, and the first look at the first room. A tree has exactly one, and it is the first card of the first segment. A `turn` card is a command the player types, given in `command`. The opening is not a turn, so the first `turn` card is turn 1.

Every card has an `id` of its own, again eight lowercase letters or digits. The identifiers mean nothing. They only have to be different from one another, and they must never change once written, because other things will come to refer to a card by its identifier.

`"skip": true` says that the card is played and nothing is checked. A card must either check something or say `skip`, and a card that does neither fails.

The layout of the file matters. The keys of every object are in alphabetical order, which is why `cards` comes before `id` and `command` before `type`. The indentation is two spaces, each element of a list is on a line of its own, one space follows each colon, and the file ends with one newline after its last brace. A tree file laid out any other way is refused.

```
test: relighting.tests/ has files not in canonical form (re-save
them from the Testing tab):
  mainline.json
```

38.4 Reading what the story says

To write an expectation you need the story's exact words. Run the tree with two options and it will give them to you.

```
sharpee test . --json --capture-output > records.txt
```

The file `records.txt` now holds one JSON record to a line. The report of derived tests that every run ends with still prints in the terminal, and you can pass over it here. The records of type `command-result` are the cards, and each carries the card's output in `actualOutput`.

```
{"schemaVersion":2,"seq":3,"elapsedMs":80,"type":"command-
result","file":"mainline","line":0,"input":"take
lamp","passed":true,"expectedFailure":false,"skipped":true,"cardId
":"card0002","actualOutput":"You take
it.","turn":1,"assertionResults":
[{"description":"[SKIP]","passed":true,"message":"Skipped"}]}
```

So the working method is this. Add the next few commands as skip cards. Run the tree, and read what each one printed. Then replace each `skip` with the part of that output you care about.

38.5 What a card can assert

A card's expectations go in `assertions`. Here are the last two cards again, with their `skip` lines replaced.

```
  {
    "assertions": {
      "exact": [
        "You take it."
      ]
    },
    "command": "take lamp",
    "id": "card0002",
    "type": "turn"
  },
  {
    "assertions": {
      "contains": [
        "In the oak press you see a library key."
      ],
      "states": [
        "the oak press is open"
      ]
    },
    "command": "open press",
    "id": "card0003",
    "type": "turn"
  }
```

There are six kinds of assertion, and a card may carry several.

- `contains` is a list of pieces of text, each of which must appear in the turn's output. This is the one you will use most.
- `notContains` is a list of pieces of text, none of which may appear.
- `exact` is the turn's whole output, line by line. Use it sparingly. It breaks when any sentence of the turn is reworded, including sentences the card was not about.
- `states` is a list of conditions, in the same words a `while` or a `when` uses, each of which must hold after the turn:

`the oak press is open`, `Tamsin Vey is bound`, `the story is awake`.

- `events` is a list of event names, each of which the turn must have raised: `media.sound.play`, or the story's own `relighting-working-learned`.
- `channels` is a list of channel claims. Each names a channel by `id` and gives either `is`, the channel's exact value, or `contains`, a list of pieces its value must include.

The opening card conventionally checks the story's title, which travels on a platform channel called `info.title`.

```
    {
      "assertions": {
        "channels": [
          {
            "id": "info.title",
            "is": "The Relighting"
          }
        ]
      },
      "id": "card0001",
      "type": "opening"
    },
```

With all three cards changed, `mainline.json` is this.

```
{
  "cards": [
    {
      "assertions": {
        "channels": [
          {
            "id": "info.title",
            "is": "The Relighting"
          }
        ]
      },
      "id": "card0001",
      "type": "opening"
    },
    {
      "assertions": {
        "exact": [
          "You take it."
        ]
      },
      "command": "take lamp",
      "id": "card0002",
      "type": "turn"
    },
    {
      "assertions": {
        "contains": [
          "In the oak press you see a library key."
        ],
        "states": [
          "the oak press is open"
        ]
      },
      "command": "open press",
      "id": "card0003",
      "type": "turn"
    }
  ],
  "id": "mainline"
}
```

Prefer `contains` and `states` together. The text says what the player was told, and the state says what became true. A turn that prints the

right sentence and changes nothing is one of the commonest defects in interactive fiction, and only a state assertion catches it.

The last two kinds are how chapters 36 and 37 are proved without a browser. The card that sparks the lamp asserts the event `media.sound.play`, and the card that reads the spark page asserts that the channel `workings` contains `spark: wakes a flame`.

38.6 Running the tree

```
sharpee test .
```

```
Loading story from: /home/ansel/relighting
Test tree: relighting.tests/ (seed 42, 1 line(s))

✓ opening-lodge
3 cards passing, 4 assertions passing
```

The first line names the folder the story was loaded from, as `play` does, and yours will name your own. Every report begins with it, and the reports quoted from here on start at the line after it.

The command exits with 0 when everything passes and 1 when anything fails. Here is the same tree after the `take lamp` card's `exact` list was replaced by `"contains": ["Taken."]`, words the story does not say.

```
Test tree: relighting.tests/ (seed 42, 1 line(s))

✗ opening-lodge — take lamp: Output does not contain "Taken."
    You take it.
2 cards passing, 3 assertions passing, 1 card failing, 1 assertion
failing
```

A failure names the line, the command, and the assertion that failed, and prints what the story said in its place. Then you decide which of the two

is wrong. If the story is, mend the story. If the story's new words are what you now want, mend the card.

38.7 A fork

So far the tree is one line. A **fork** is a card with more than one continuation, and it is where a tree earns its name.

A fork is always the last card of its segment. Each way on from it is a segment of its own, which names the segment it grows from in `parent` and takes a number in `ordinal`. The continuation of the same line is ordinal 0. The alternatives are 1, 2, 3 and so on.

After `open press`, the main line goes on to take the key.

```
{
  "cards": [
    {
      "assertions": {
        "contains": [
          "You take the library key from the oak press.",
          "Hobb closes the oak press."
        ],
        "states": [
          "the oak press is closed"
        ]
      },
      "command": "take key",
      "id": "card0004",
      "type": "turn"
    }
  ],
  "id": "onward01",
  "ordinal": 0,
  "parent": "mainline"
}
```

A second segment tries something else from the same position.

```
{
  "cards": [
    {
      "assertions": {
        "contains": [
          "You find you cannot."
        ]
      },
      "command": "eat cake",
      "id": "card0005",
      "type": "turn"
    },
    {
      "assertions": {
        "notContains": [
          "library key"
        ]
      },
      "command": "inventory",
      "id": "card0006",
      "type": "turn"
    }
  ],
  "id": "thecake1",
  "ordinal": 1,
  "parent": "mainline"
}
```

```
Test tree: relighting.tests/ (seed 42, 2 line(s))

✓ opening-lodge
✓ lodge · eat cake
6 cards passing, 9 assertions passing
5 commands (5 authored + 0 replayed), 1 restored at a fork
```

The tree now has two **lines**. A line is one path from the opening to an end, and the report names each by the room it starts in and its first command. The main line is named for the opening.

The second line did not replay the first three cards. The runner saved the story at the fork and restored it, so a deep tree with many branches costs little more to run than its cards.

The second card of the branch is worth a look. `inventory`, asserting that the key is not there, proves that this line really did start from before the key was taken. A branch is a different world from its siblings, and nothing done in one is seen in another.

38.8 An ending

A turn that ends the story says so on its card, with the ending's key from chapter 18.

```
{
  "assertions": {
    "contains": [
      "The two lights hold one another.",
      "You have risen to the rank of a Lamplighter."
    ],
    "events": [
      "media.image.hide"
    ]
  },
  "command": "spark beacon",
  "ending": "beacon-lit",
  "id": "bjhmo3xt",
  "type": "turn"
}
```

A card with `ending` asserts that the story ended on that turn, and in that ending. It is the last card of its line.

38.9 The whole tree, and what it covers

The companion folder `chapter-38` holds the story as chapter 37 left it and the tree this book has been running since chapter 5, grown to its full

size: 182 cards on seven lines. The main line is the winning walk. The six branches are the bargain refused and reopened, Tamsin Vey at the grille, the pry bar heard from the Library, the cracked lens and its ending, the parapet and its ending, and Hobb's trust.

```
Test tree: relighting.tests/ (seed 42, 7 line(s))

✓ opening-lodge
✓ refectory · no
✓ library · east
✓ library · wait
✓ undercroft · put lens in hoist
✓ undercroft · drop lens
✓ undercroft · up
182 cards passing, 275 assertions passing
181 commands (181 authored + 0 replayed), 6 restored at a fork
```

After the cards, the report goes on at length, and the rest of it is not about your tree. `sharpee test` also reads the story itself, finds every clause and every branch inside a clause, and tries to write a small test for each one unaided: arrange the world so that the clause applies, type the command, and see that the clause's effects happened. These are the **derived tests**. A `✓` is a clause that was exercised that way. A `◌` is one the runner could not arrange, with its reason. A `✗` is a clause whose effects did not happen, and it fails the run as a card does.

The report closes with three counts. Under the first it lists every branch not exercised, 81 lines here, which this page leaves out.

```
Branches exercised: 53 / 134
```

```
Endings reached: 2 / 3
Not reached (1):
  relighting.story:314 · night-falls (lose)
Rooms entered: 13 / 13
```

Read the counts as a list of what nobody has checked. The Relighting has three endings and the tree reaches two. The third, `night-falls`, needs a

hundred and sixty turns of doing nothing, and it has been left out on purpose so that this report has something to show you. In your own story, an ending in that list is an ending no test has ever seen.

38.10 Build and try it

Make the folder `relighting.tests` beside your story. Write the manifest into it, and `mainline.json` with its three skip cards, with their keys in alphabetical order and two spaces of indentation for each level. Run the command of “Reading what the story says”. Then replace `mainline.json` with the whole file printed under “What a card can assert”, and run `sharpee test .`. Write the other two segments, `onward01.json` and `thecake1.json`, and run it again. The second and third segments go in together, because a segment that continues a line is refused until a branch leaves the card it continues from. Then break something on purpose. In the story, find the phrase `cake-watched`, which the seed cake’s refusal prints, and change its last sentence from “You find you cannot.” to “You decide against it.” Run the tree and read the failure. Then put the sentence back and run the tree again.

When you have seen a failure and mended it, replace your `relighting.tests` folder with the `relighting.tests` folder from the companion folder `chapter-38`, and run the full tree. Your story stays as it is.

```
sharpee test .
```

UNDER THE HOOD: THE TREE RUNNER · MANIFEST `SEED`, CARD `STATES`, PLATFORM CHANNEL `INFO.TITLE`

The runner loads the compiled story exactly as `sharpee play` does, sets the story's random seed from the manifest, and feeds each card's command to the real parser and the real turn loop. A `contains` assertion reads the turn's prose. A `states` assertion is compiled as a Chord condition and evaluated against the world after the turn. An `events` assertion reads the turn's event list, and a `channels` assertion reads the values the turn would have sent to a page. Nothing is simulated, which is why a tree that passes is evidence about the story a player will get.

FROM ANOTHER SYSTEM: TEST COMMANDS AND SCRIPTS

Inform 10 lets a story name a list of commands in its source, as in `Test balloon with "get balloon / blow balloon / drop balloon".`, and typing TEST BALLOON in the running story plays them. *Writing with Inform* adds that "The Skein and Transcript tools of the Inform application are provided for exactly this purpose". TADS 3 has input scripts: its System Manual says "The Interpreter provides"script" files for this purpose", read at start with the interpreter's `-i` option, and a game built on the standard library offers RECORD and REPLAY commands. Each of these replays commands. What a Chord card adds is the expectation stored beside the command, and the fork.

38.11 What you have so far

You can write a play-through down as cards, say on each card what the story must print and what must have become true, fork it wherever the

player has a choice worth checking, and run the whole thing in a few seconds after every change.

A tree proves what happens along the paths you recorded. It does not, by itself, say which of those results you are promising to keep. The next chapter is about writing the promises down.

39 Claims

A test tree checks a great many small things. Card by card it says that this sentence was printed and that door was open. Nowhere does it say, in one place and in plain words, what the story is for: that the stair can be opened, and that the game can be won.

A **claim** is one sentence of that kind, with a name. The claims of a story live together in a small file of their own, and `sharpee test` reports on every one of them each time it runs. This chapter writes three claims for The Relighting, and looks at what the report says when a claim holds, when it stops holding, and when nobody can tell.

39.1 A fact worth keeping

The quickest way to a claim is to take one from a line of the tree that already reaches it. The `sharpee claim` command does that. Give it a line and it lists what is true at the end of that line that was not true when the story began.

A line is named by the identifier of the segment it starts in. If you name one that does not exist, the command lists the ones that do.

```
claim: no line "winning1" — the tree's lines, by segment id and
first command:
  thelodge  talk to hobb
  refusal1  no
  grille01  east
  fromafar  wait
  cracked1  put lens in hoist
  parapet1  drop lens
  trusted1  up
```

The main line is `thelodge`. The option `--at` chooses a turn on it, and without the option the command reads the line's last card. The turn of a

card is the `turn` field of its record in the output of chapter 38's `sharpee test . --json --capture-output`. Two cards on the way type `unbind grille`. The one at turn 70 is refused, because the bell is tolling. The one at turn 82 lifts the grille, and it is the one wanted here.

```
sharpee claim thelodge --at 82
```

```
At turn 82 of line thelodge "opening-lodge", true now and not at
boot:
  the player is in the Bell Stair
  the player has the spectacles
  the player has the hand lamp
  the player has the library key
  the player has the unbind page
  the player has the spark page
  the player has the hush page
  the story is waking
  the iron grille is lifted
  the spark page is learned
```

The list goes on for another twenty-three lines, and it closes by saying how to keep one.

```
Lock one with: sharpee claim thelodge "<fact>" --at 82
```

Every line of the list is a sentence a claim can state. Most of them are not worth stating. That the player has the spectacles at turn 82 is true, and nobody would mind if it stopped being true. The line that matters is `the iron grille is lifted`, because the whole middle of the story exists to make it so.

39.2 Locking it

Give the command the fact, in quotation marks, and a name for the claim.

```
sharpee claim thelodge "the iron grille is lifted" --at 82 --name
↪ "the ward on the stair can be lifted"
```

```
Locking at turn 82 of line thelodge "opening-lodge": the iron
grille is lifted

  needs rooms: the Lodge, the Gate Court, the Cloister, the
Refectory, the Library, the Bell Stair
  needs things: Hobb, the kettle, the chair, the spectacles, the
hand lamp, the oak press, the library key, Tamsin Vey, the unbind
page, the library door, the grimoire, the spark page, the hush
page, the mend page, the wisp, the wax taper, the tower bell, the
newspapers, the candles, the iron grille
  needs verbs: talk to, ask, tell, goodbye, sit on, exit, search,
take, examine, open, wait, say, answer, unlock, consult, look up,
read, light, spark, look, inventory, unbind, quiet, hush

Written: created relighting.claims.chord
Written: relighting.tests/manifest.json
Written: relighting.tests/thebell1.json
Pinned "the ward on the stair can be lifted" on card b7wicd1b.
```

Three files changed. The command wrote a new file beside the story, `relighting.claims.chord`, which holds the claim. It added one key to the tree's manifest, so that the tree knows the file is there.

```
{
  "claims": "relighting.claims.chord",
  "seed": 42,
  "story": "relighting",
  "version": 3
}
```

And it added one key, `locks`, to the card where the fact was read.

```
    {
      "assertions": {
        "contains": [
          "The grille is not there, in the manner of a thing that
          ↪ was never there"
        ],
        "states": [
          "the iron grille is lifted"
        ]
      },
      "command": "unbind grille",
      "id": "b7wicd1b",
      "locks": [
        "the ward on the stair can be lifted"
      ],
      "type": "turn"
    },
```

A **lock** ties a claim to a card. It says that the claim is true on that card, on that turn of that line, and that you want to be told if it ever is not.

Add `--propose` to the command and it prints the same report and writes nothing, which is the way to look before you lock.

39.3 The claims file

Here is the file the command wrote.

```
## The Relighting's claims: what the finished story can reach,
↪ each under the
## rooms, things and verbs that are load-bearing on the way. The
↪ test tree's
## manifest names this file; the story never imports it.

claims
  claim the ward on the stair can be lifted
    the iron grille is lifted
    needs rooms: the Lodge, the Gate Court, the Cloister, the
    ↪ Refectory, the Library, the Bell Stair
    needs things: Hobb, the kettle, the chair, the spectacles, the
    ↪ hand lamp, the oak press, the library key, Tamsin Vey, the
    ↪ unbind page, the library door, the grimoire, the spark page,
    ↪ the hush page, the mend page, the wisp, the wax taper, the
    ↪ tower bell, the newspapers, the candles, the iron grille
    needs verbs: talk to, ask, tell, goodbye, sit on, exit,
    ↪ search, take, examine, open, wait, say, answer, unlock,
    ↪ consult, look up, read, light, spark, look, inventory,
    ↪ unbind, quiet, hush
end claims
```

The file is Chord, and it holds one block, which opens with `claims` and closes with `end claims`. It has no story header, and the story does not import it. The manifest's `claims` key is the only thing that points to it, so a story is never changed by the claims made about it.

Inside the block, each claim begins with the word `claim` and its name. The name is yours, and it is what the report will call the claim. Write it as you would say it to someone: the ward on the stair can be lifted.

The next line is the fact, and a claim has exactly one. Under it come up to three `needs` lines, which this chapter returns to when it reaches the walk. For a claim that a line of the tree reaches, they are a note of what the line used on the way.

39.4 What a claim may state

A claim states one thing that holds. The forms are few, and most of them are conditions you already write after `while` and `when`.

- `the iron grille is lifted` names a thing and one of its states. The same form takes the platform's own state words: `open` and `closed`, `locked` and `unlocked`, `on` and `off`, `lit` and `unlit`, `worn` and `unworn`, `read` and `unread`.
- `the story is awake` names a state of the story itself.
- `the player is in the Library` names the room the player has reached.
- `the player has the library key` says who holds a thing. Any person may stand where `the player` does.
- `the lens is in the Great Lantern` says where a thing is, and `the kettle is not in the Lodge` names a room that a thing has left.

Three more forms belong to claims alone.

- `the story ends in victory` and `the story ends in defeat` say how the story ended, and `the story has ended` says only that it did.
- `Hobb was asked about "the kettle"` says that a person was asked about a topic.
- `the seed cake has been eaten` says that a thing has answered an action. The thing must have an `on` clause for that action, and the word after `has been` is the action's past participle.

A claim cannot be turned round. There is no `is not lifted` and no `never`, because a claim says what the story can reach, and "can never reach" is a different and much harder thing to show.

When a fact does not fit the story, `sharpee test` refuses the file before it runs anything, and says what the thing's states are.

```
test: claims file 'relighting.claims.chord' does not compile (1
error(s)):
  relighting.claims.chord:7:24 [claims.state-unknown]
[relighting.claims.chord] `the ward on the stair can be lifted`:
`iron grille` is never `raised`. Its states are `standing`,
`lifted`. The platform's words are `open`, `closed`, `locked`,
`unlocked`, `on`, `off`, `lit`, `unlit`, `worn`, `unworn`, `read`,
`unread`.
```

39.5 The report

Run the tree as before.

```
sharpee test .
```

After the cards, the derived tests and the three counts of chapter 38, the report has a new last section.

```
Claims: relighting.claims.chord
claims: 1 of 1 held (seed 42)
0 walks
HELD  the ward on the stair can be lifted — by line thelodge
"opening-lodge" at turn 82
  lock "the ward on the stair can be lifted" at line thelodge
"opening-lodge" at turn 82 — held there
```

The first line under the file's name is the score: how many of the claims you have declared are held. A claim is **held** when the runner has seen the story reach it. Here a line of the tree reached it, at turn 82, and the report says which.

The last line is the lock. The claim is held and its lock is held, and those are two different statements. The first says that some line, somewhere, reaches the fact. The second says that the fact is true on the very card you pinned it to.

39.6 When a lock breaks

Suppose that some weeks from now you are tidying the `ward` trait, and the line `change it to lifted` goes out with something else. The grille still vanishes when it is unbound, because `move it offstage` is still there, and the prose still prints. Run the tree.

```
✗ opening-lodge — unbind grille: State assertion failed: the iron
grille is lifted. the iron grille is "standing", expected "lifted"
    You say the word. The grille is not there, in the manner of a
thing that was never there, and the stair goes up.
```

The card fails, as chapter 38 promised. Thirty-four more cards fail after it, on the main line and on the two branches that climb the stair, because the stair is still shut and everything beyond it goes wrong. Thirty-five failures do not tell you much about what you broke. The claims section does.

```
Claims: relighting.claims.chord
claims: 0 of 1 held, 1 unproven (seed 42)
1 walk
  under the set of "the ward on the stair can be lifted": 6 rooms,
20 things, 24 verbs — 11612 states, 63603 commands, stopped: max-
seconds
UNPROVEN  the ward on the stair can be lifted — not reached by any
line; the walk stopped at max-seconds after 11612 states; its set
carries 20 things, and each independent thing doubles the states
to exhaust
  lock "the ward on the stair can be lifted" at line thelodge
"opening-lodge" at turn 82 — NOT held there: the iron grille is
standing
```

Read it from the bottom. The lock is not held, and the report says what is so in its place: the iron grille is standing. One line names the promise you broke, in the words you gave it.

The line above it says that the claim is **unproven**. No line of the tree reaches the fact any longer, so the runner went looking for a way to reach

it by itself, and gave up after two minutes. The next section is about that search.

The two are treated differently. A lock that is not held fails its card, as a failed assertion does, and the run exits with 1. Had this card carried no `states` assertion, it would have failed all the same, with a line of its own.

```
✗ opening-lodge — hush bell: lock "the ward on the stair can be
lifted" not held here: the iron grille is standing
```

(That line comes from a lock pinned, on purpose, one card too early.)

An unproven claim fails nothing. Claims are a list of what you mean the story to do, and the report on them is a reading of how much of the list is true today. That makes it safe to write a claim before the story can meet it.

Put the line back, and the report returns to 1 of 1.

39.7 A claim written by hand

The second claim is the one any story should make.

`sharpee claim` proposes a set of needs from what the line did, and the proposal is a draft. It lists everything the line touched, the kettle and the chair among them. It is also not yet sound for every line. At this edition the command does not return when it is asked to lock a fact at a turn that comes after the line's first `put`, and on the winning line that is every turn from 105 on. The winning turn is 138.

So this claim is written by hand, which is always allowed. A claims file is yours to edit. Add the block before `end claims`, with one blank line above it.

```
claim the story can be won
  the story ends in victory
  needs rooms: the Lodge, the Gate Court, the Cloister, the
  ↪ Refectory, the Library, the Bell Stair, the Undercroft, the
  ↪ Lantern Gallery, the Water Stair, the Beacon Isle
  needs things: Hobb, the spectacles, the hand lamp, the oak
  ↪ press, the library key, Tamsin Vey, the unbind page, the
  ↪ library door, the spark page, the hush page, the mend page,
  ↪ the tower bell, the newspapers, the candles, the iron
  ↪ grille, the cistern, the crate, the lens, the book hoist,
  ↪ the shutter crank, the shutter, the Great Lantern, the Mere,
  ↪ the skiff, the beacon
  needs verbs: talk to, ask, say, take, open, unlock, read,
  ↪ light, spark, examine, wait, unbind, quiet, fix, repair,
  ↪ turn, push, enter, steer, make sail, exit
```

The verbs leave out `put`, although the winning line types it twice. At this edition a `needs verbs` line that names `put` stops `sharpee test` from finishing.

The lock is one key on the winning card, which is the card `bjhmo3xt` in `winning1.json`. It is written between `id` and `type` because a tree file keeps its keys in alphabetical order.

```
    "command": "spark beacon",
    "ending": "beacon-lit",
    "id": "bjhmo3xt",
    "locks": [
      "the story can be won"
    ],
    "type": "turn"
```

The layout of the claims file matters as the layout of a tree file does. A claim's name is indented by two spaces and its lines by four. The fact comes first, then `needs rooms`, `needs things` and `needs verbs` in that order, and exactly one blank line separates two claims. A file laid out another way is refused, and the refusal names the line. This is the refusal for a file in which a second claim follows this one with no blank line between them.

```
test: claims file 'relighting.claims.chord' is not in the lock's
canonical form (fix the lines named, or re-lock):
  relighting.claims.chord:17 · claim the story can be won — the
next block follows without a blank line
```

With the file laid out as the runner wants it, the report has both claims.

```
claims: 2 of 2 held (seed 42)
0 walks
HELD  the ward on the stair can be lifted — by line thelodge
"opening-lodge" at turn 82
HELD  the story can be won — by line thelodge "opening-lodge" at
turn 138
  lock "the ward on the stair can be lifted" at line thelodge
"opening-lodge" at turn 82 — held there
  lock "the story can be won" at line thelodge "opening-lodge" at
turn 138 — held there
```

39.8 A claim no line reaches

Both claims so far were read from the tree. A claim may also say something that no line of the tree does, and then the runner has to find out for itself whether it is true.

Hobb's seed cake has sat on his chair since Part II, and since Part III the player has been refused it: "Hobb cut that cake into two pieces with some care, and he is sitting four feet away. You find you cannot." The refusal has a condition, `when Hobb is here`. So the cake can be eaten, somewhere Hobb is not. Nothing in the tree tries it.

```
  claim the cake can be eaten in the Lodge
    the seed cake has been eaten
    needs rooms: the Lodge
    needs things: Hobb, the chair, the seed cake
    needs verbs: take, eat

  claim the cake can be eaten
    the seed cake has been eaten
    needs rooms: the Lodge, the Gate Court
    needs things: Hobb, the chair, the seed cake
    needs verbs: take, eat
```

The two claims state the same fact and differ in one word of one line. The first allows the Lodge and nothing else. The second allows the Gate Court as well.

```
claims: 3 of 4 held, 1 unproven (seed 42)
2 walks
  under the set of "the cake can be eaten in the Lodge": 1 rooms,
3 things, 2 verbs — 161 states, 239 commands, stopped: max-depth-
reached
  under the set of "the cake can be eaten": 2 rooms, 3 things, 2
verbs — 17 states, 20 commands, stopped: all-claims-settled
HELD  the ward on the stair can be lifted — by line thelodge
"opening-lodge" at turn 82
HELD  the story can be won — by line thelodge "opening-lodge" at
turn 138
UNPROVEN  the cake can be eaten in the Lodge — not reached by any
line; the walk stopped at max-depth-reached after 161 states; its
set carries 3 things, and each independent thing doubles the
states to exhaust
HELD  the cake can be eaten — not reached by any line; by walk at
depth 3: take seed cake / north / eat seed cake
```

For each claim that no line reaches, the runner takes a **walk**. It starts the story fresh and tries commands, and from every position those lead to it tries commands again, keeping count of the positions it has seen. It stops when it finds one where the claim is true.

For the second claim it found one in twenty commands, and the report prints the way there: take the cake, go north, eat it. Those three commands are a **witness**. The claim is held, and it is held more firmly than by a line of the tree, because nobody wrote the answer down.

39.9 The needs lines

A walk that tried every command in every room would never finish. The `needs` lines are what make it small. They say which rooms, things and verbs are on the way to the fact, and the walk stays inside them.

- `needs rooms` lists the rooms the walk may enter. It goes through an exit only when the room beyond is on the list.
- `needs things` lists the things and people a command may name. The player is always there and is never listed.
- `needs verbs` lists the verbs the walk may type, as a player types them: `take`, `eat`, `talk to`.

The walk does not invent sentences. From each position it tries the exits, and for each thing it can see it tries what that thing's kind allows: `open` for something openable, `take` for something that can be carried, `ask` for a person with topics, a story's own verb for a thing that answers it. Then the three lists cut that down.

The lists are part of the claim. The first cake claim does not say that the cake can be eaten. It says that the cake can be eaten using only the Lodge, Hobb, the chair, the cake, and the verbs `take` and `eat`. That is a smaller statement, and in this story it is false, because Hobb never leaves the Lodge. The walk tried 161 positions, reached the deepest path it allows itself, and stopped.

It did not report the claim as false. **Unproven** means that the walk ran out of room before it found a witness. Sometimes that is because there is none, as here. Sometimes the set is too loose, and the walk drowned in kettles and chairs. The report cannot tell the two apart, and it says which limit stopped it so that you can judge: `max-depth-reached` is a path of

eighty commands, `max-states` is half a million positions, and `max-seconds` is two minutes. Run the tree with `--verbose` and the limits are printed.

```
Claims budget per walk: 120s, 500000 states, depth 80
```

The last clause of an unproven line is advice. Every thing in the set that can change by itself doubles the number of positions there are to visit. A walk under three things is over in a moment. The walk under the first claim's twenty things, in the section on the broken lock, ran for its whole two minutes. So when a claim comes back unproven, make its set smaller before you make it larger, and list only what the fact cannot be reached without.

When several claims need the same set, the lists can be written once, at the top of the block, before the first claim. A claim with no `needs` lines of its own takes those, and a claim that writes one of the three replaces that one and keeps the other two.

```
claims
  needs rooms: the Lodge, the Gate Court
  needs things: Hobb, the chair, the seed cake
  needs verbs: take, eat

  claim the cake can be had
    the player has the seed cake

  claim the cake can be eaten
    the seed cake has been eaten

  claim the cake can be eaten in the Lodge
    the seed cake has been eaten
    needs rooms: the Lodge
end claims
```

Claims that share a set share a walk, so the first two here cost one walk between them.

39.10 Which claims to keep

The first cake claim has done its work in this chapter, and it comes out of the file. An unproven claim that you know to be false is only noise in the report. An unproven claim that you believe is a different matter. It is the most useful line in the file, because it is a thing the story does not yet do.

That is the other way to use claims. Before a puzzle exists, write down that it can be solved: name the fact, and list the rooms and things you intend to build it from. The claim is unproven on the first day. On the day the puzzle works, the walk finds its witness, and you have not had to play it through to know.

Keep the claims few. The Relighting has three: the stair opens, the story can be won, and a small one about a cake that shows the walk at work. A story of this size might fairly have a dozen. They should be the sentences you would use to tell someone what the game is.

39.11 Build and try it

Work in your own project, which now holds what the companion folder `chapter-38` holds: the story of chapter 37 and the full tree. List the facts at turn 82 of the main line, and lock `the iron grille is lifted` under the name this chapter gave it, `the ward on the stair can be lifted`, so that your reports read as the chapter's do. Run the tree and find the claim and its lock in the report. Add the claim that the story can be won, and its lock, as the chapter did.

Then add the second cake claim by hand, and run the tree again to see the witness. Change its `needs rooms` line to the Lodge alone, and see it come back unproven. Put the Gate Court back on the line afterwards.

Last, break the story as this chapter did, by taking `change it to lifted` out of the `ward` trait, and read the lock's line. The chapter broke it with one claim in the file. With three, the claims section is longer than the one printed under "When a lock breaks": the

claim that the story can be won is unproven as well, its lock is not held either, and the runner walks for each of them, so the run takes four minutes at the least, and a good deal longer on a busy machine. Put the line back.

The companion folder `chapter-39` holds the story with its three claims.

```
sharpee test .
```

```
Claims: relighting.claims.chord
claims: 3 of 3 held (seed 42)
1 walk
  under the set of "the cake can be eaten": 2 rooms, 3 things, 2
verbs — 17 states, 20 commands, stopped: all-claims-settled
HELD  the ward on the stair can be lifted — by line thelodge
"opening-lodge" at turn 82
HELD  the story can be won — by line thelodge "opening-lodge" at
turn 138
HELD  the cake can be eaten — not reached by any line; by walk at
depth 3: take seed cake / north / eat seed cake
  lock "the ward on the stair can be lifted" at line thelodge
"opening-lodge" at turn 82 — held there
  lock "the story can be won" at line thelodge "opening-lodge" at
turn 138 — held there
```

UNDER THE HOOD: THE CLAIMS TIER · MANIFEST `claims`, CARD `locks`, VERDICTS `held` AND `unproven`

`sharpee test` runs in three tiers: the tree's cards, the derived tests, and the claims. The claims file is compiled against the story before any card is played, which is why a misspelled state stops the run at once. While the cards replay, every claim is checked after every card. A claim that some card satisfies is held by that line and is never walked, and a card's `locks` are checked on that card alone. Only the claims left over are walked, one walk for each distinct set of needs, each from a fresh start at the tree's seed. Every line of the claims report is the same on every run at the same seed, except where a walk was stopped by the clock.

FROM ANOTHER SYSTEM: PROVING A STORY CAN BE FINISHED

Inform 10 and TADS 3 both replay commands, as the last chapter's aside describes, and a replayed solution is how an author in either system shows that a story can still be won. A lock is the same idea with a name on it. The walk has no counterpart in the pages of *Writing with Inform* and the TADS 3 System Manual read for this book: neither describes a tool that searches a story for a way to reach a state. The nearest thing is in *Writing with Inform*'s section on testing for thoroughness, which names an extension that "allows us to try out a long list of commands against any object in the story". That tries every command on one thing. It does not look for a path.

39.12 What you have so far

You can write down, in a file beside the story, the few sentences the story exists to make true. You can lock one to the card where a line of the tree reaches it, and let the runner go looking for the ones no line reaches. The report tells you after every change how many are held.

The Relighting is written, tested, and holds its claims. The last chapter gives it a version and an address, and sends it out.

40 Publishing

The Relighting has been a folder on your machine for thirty-nine chapters. This one turns it into a single file that you can give to someone who has never heard of Chord, and who will open it in a browser and play.

There is little left to write. Publishing is mostly a matter of deciding what the story is called for good, what number it carries, and whether its source goes with it.

40.1 The identifier

Chapter 2 mentioned a header field called `ifid:` and said that you do not need to write one. Chapter 37 built the page for the first time, and the build's first line was this.

```
  ✓ Minted the story's ifid into relighting.config.json
```

An **IFID** is an Interactive Fiction Identifier: a long code that belongs to one work of interactive fiction and to no other. Archives and catalogues file a story under it. The title of a story can change, and two stories can share a title. The identifier is what stays the same from the first release to the last.

The first build made one, and wrote it to a small file beside the story.

```
{
  "ifid": "C24E29F8-C36B-4F55-9FE9-B6D75AE7133D",
  "version": 1
}
```

That file, `relighting.config.json`, is where the identifier lives. The `version` in it is the version of the file's own layout and has nothing to

do with your story. The build also wrote a line into the story's header, under `id:`.

```
story
  title: The Relighting
  id: relighting
  ifid: C24E29F8-C36B-4F55-9FE9-B6D75AE7133D
```

The header's line is a copy, put there so that you can see the identifier where you see the title. The file is the original. If the two ever differ, the next build writes the file's value back into the header, and says nothing about it.

Three rules follow, and they are the whole of what you need to know.

Keep `relighting.config.json` with the story. If you keep your story under version control, the file goes in. If you copy the story to another machine, the file is copied too.

Never make a second identifier for the same story. A new release of The Relighting is still The Relighting.

Do make a new one for a different story. If you copy this folder to begin a story of your own, delete the config file and the header's `ifid:` line from the copy, and let its first build mint a fresh one.

The command line has a small tool for the rare day when you need an identifier by itself.

```
sharpee ifid generate
```

```
C6ABD976-A999-4F4F-92A8-B393F82127A6
```

40.2 The version

`story-version:` has stood at `0.7.0` since Part VII. The number is yours to count as you please, and the player sees it. The published page prints it under the title when it opens. The terminal prints the same four lines with the version last.

```
The Relighting
Story v1.0.0
Thirty years after the magic went out of the world, a lamp burns
blue.
By The Sharpee Project
```

A common habit is to keep the first number at 0 while the story is unfinished, and to make it 1 on the day you first hand the story to strangers. Today is that day.

```
story-version: 1.0.0
```

After that, raise the last number when you mend something, and the middle number when you add something. The reason to be careful about it is practical. When a player writes to say that the grille will not lift, the first thing you will ask is which version they are playing.

The identifier and the version answer different questions. The identifier says which story this is, and it never changes. The version says which release of it, and it changes every time.

40.3 Whether the source goes too

By default a published story carries no source. The page holds the compiled story, and your `.story` file stays at home. One header line changes that.

```
publish-source: yes
```

The line goes in the story header with the other fields, and its place among them does not matter. With the line, the `.story` file is copied into the published folder under its own name, and the fragments it imports go with it. The Relighting is a teaching story and it takes the line. For your own story the choice is yours, and leaving the line out is a perfectly good answer.

40.4 Publish

Before you publish, run the tree one last time and read the claims. A story that is about to go to strangers should hold every claim you made for it.

```
sharpee test .
```

```
claims: 3 of 3 held (seed 42)
```

Then publish.

```
sharpee publish relighting.story
```

```
📦 Publishing relighting

  Story: The Relighting (relighting)
  ✓ Validated relighting.story (gate-clean) and shipped its source
(`publish-source: yes`)
  ✓ Shipped 1 import fragment(s) → imports.json
  ✓ Story IR → dist/relighting.ir.json
  Generated browser entry from template
  ✓ Embedded the compiled story IR into the bundle
  Bundling game.js...

  dist/web/relighting/game.js  1.9mb ⚠

⚡ Done in 66ms
  ✓ Built game.js
  ✓ Copied index.html
  ✓ Copied index-testing.html (IDE testing surface — excluded from
publish)
  ✓ Copied platform engine CSS (base, engine, decorations)
  ✓ Wired 2 theme(s): paper, modern-dark
  ✓ Copied relighting.css
  ✓ Copied assets/ (2 entries)

✅ Build complete! (game.js 1907.7 KB)
Output: dist/web/relighting/

  IFID:   C24E29F8-C36B-4F55-9FE9-B6D75AE7133D
  Output: dist/relighting.zip (0.7 MB)

  Unzip anywhere and open index.html, or upload the zip to
itch.io.
```

Most of that is the build of chapter 37, and `publish` does run a build. It differs from `sharpee build` in four ways.

It empties `dist/web/relighting` first, so that nothing left over from an earlier build can travel with the story.

It leaves out the source map, the file `game.js.map` that chapter 37's build wrote beside `game.js`. A source map is for finding faults, and a player has no use for it.

It refuses a story that has no identifier. A build will mint one. A publish never will, because publishing is the one moment when inventing a second identifier for an existing story would do real harm.

```
publish: relighting.story has no identity — no story config and no
`ifid:` header. If this story ever had one, restore its `<story-
name>.config.json` from version control; otherwise run `sharpee
build` once (or open the story in Chord Writer) to mint it.
```

And it writes the zip.

40.5 What ships

`dist/relighting.zip` is the whole published story, in fourteen files.

```
audio/catch.mp3
audio/lapping.mp3
audio/toll.mp3
base.css
decorations.css
engine.css
game.js
img/lantern-lit.png
imports.json
index.html
relighting.css
relighting.story
themes/modern-dark.css
themes/paper.css
```

`index.html` is the page, and it is at the top of the archive, where a site that hosts web games expects to find it. `game.js` is the story and everything needed to run it. The three platform stylesheets, the two themes from the header's `themes:` line, and the story's own `relighting.css` follow. The `audio` and `img` folders are the `assets` folder of chapter 36. `relighting.story` and `imports.json` are there because of `publish-source: yes`, and without that line the archive has twelve files.

Look also at what is not there. The test tree is not in it, and neither is the claims file. Those are your instruments, and they stay with you. The page `index-testing.html`, which the build makes for Chord Writer, is left out as the report says.

The command's last line tells you what to do with the zip. Anyone who unzips it has the folder that chapter 37 served with `npx serve`. Any place that will hand out files will hold it: a folder on a website of your own, or a site that hosts web games and takes a zip with an `index.html` at its top.

Two options are worth knowing. `--out` followed by a path writes the zip somewhere other than `dist`. `--no-menu` publishes a page without the menu that carries Save, Restore, Restart and Quit, for an author whose own page supplies them.

40.6 After the first release

A story is seldom finished on the day it is published. Somebody will find the thing you missed.

When they do, mend the story, and add a card to the tree that would have caught the fault. Raise `story-version:`. Run `sharpee test .` and read the claims. Run `sharpee publish` again and put the new zip where the old one was. The identifier does not change, so the archives know it for the same story, and the version tells the two releases apart.

40.7 Build and try it

Set `story-version:` to `1.0.0` in your own copy. The Relighting ships its source, so write `publish-source: yes` in the header; the report and the list of files in this chapter are those of a story that has the line. Run the tree. Then publish, and unzip `dist/relighting.zip` into an empty folder. Any unzipping tool will do. From a terminal in the project folder:

```
unzip dist/relighting.zip -d ../relighting-published
```

Serve that folder with `npx serve .` from inside it, and play the story you wrote from the lodge to the beacon, as a stranger will.

```
sharpee publish relighting.story
```

The companion folder `chapter-40` holds the story as it was published for this book.

> **UNDER THE HOOD: IDENTITY AND THE ARTIFACT · `RELIGHTING.CONFIG.JSON`, `DIST/RELIGHTING.ZIP`**
>
> The story config is the tool's own file. A build creates it when it is missing and copies its identifier into the header, and nothing else in the toolchain writes to it: `compose`, `test` and `play` only read the story. `publish` runs the same compile gate as `compose --check`, then the browser build with the source map turned off, and then zips `dist/web/<id>` with `index.html` at the root of the archive. The compiled story is the Story IR of chapter 2, embedded in `game.js`, and a copy is left at `dist/relighting.ir.json` for tools to read.

FROM ANOTHER SYSTEM: IDENTIFIERS AND RELEASES

The IFID is common ground. *Writing with Inform* says that "each new project created by Inform is allocated a unique identification number called its IFID", and that "The IFID is the equivalent for IF of the ISBN of a printed book." A TADS 3 game states its own in its `versionInfo` object, in a line such as `IFID = '...'`, beside `version = '1.0'`. Inform 10 numbers releases with a sentence, `The release number is 7.`, where Chord has `story-version:`.

For the artifact, Inform 10 compiles a story file, and a `Release along with` sentence lists what goes out beside it. The list may include `a website`, and `Release along with an interpreter.` adds what *Writing with Inform* calls "A playable web page". TADS 3 compiles source into an image file, which its System Manual calls "the T3 executable file that you run using the TADS 3 interpreter". Chord has no story file and no separate interpreter. The published page is the only artifact, and the interpreter is inside it.

40.8 What you have

You have a story with an identifier that will follow it for good, a version that says which release a player holds, and a zip that plays in any browser.

That is the whole of the road this book set out on. You began with a header and one room. You have since written rooms and the things in them, states and the clauses that change them, phrases, new verbs, people who talk and act for themselves, a boat and a beacon, sound and a page to hear it on, a tree that plays the story for you, and three sentences that say what it is for.

The appendices that follow are for looking things up: the grammar, the kinds and trait words, the standard actions, and the rest. The next story is yours.

Appendix A. The grammar

The Chord grammar in EBNF, as the compiler implements it: 215 rules in 24 sections. The grammar file's comments are left out, so this is the productions alone. Generated from `chord.ebnf` (Sharpee 5.7.0 · Chord 4.1.0) by `scripts/generate-chord-appendices.cjs`.

```
Notation:
  UPPER      = token class          "quoted" = literal word
  [ x ]      = optional             { x }    = zero or more
  x | y      = alternatives         ( * … * ) = comment (written
without the spaces)
  NL         = end of source line (Chord is line-oriented)
  >>>        = an indented block follows (layout; EBNF cannot
express
               indentation — the summary below is the rule)

Layout summary:
  - Indentation is spaces only; a tab is a lex error.
  - Blocks open with their header line and close at dedent or
their
    explicit "end <kw>" terminator per the Layout table.
  - A blank line inside a prose block is a paragraph break; in
create
    blocks a blank line ends the block, but later bare paragraphs
    append to the description.
```

Tokens

```
WORD    = letter { letter | digit | "'" | "-" | "_" } ;
NUMBER  = digit { digit } { "." digit { digit } } ;
STRING  = '"' any-except-'"' '"' ;
MARKER  = "{" content "}" ;
LOCALE  = WORD ;
rest-of-line    = any-except-NL ;
prose-line      = rest-of-line NL ;
prose-paragraph = prose-line { prose-line } ;
inline-prose    = >>> prose-paragraph ;
```

Comments

```
comment-line = "##" rest-of-line NL ;
comment-run  = comment-line { comment-line } ;
```

Top level

```
story-file   = [ comment-run ] [ story-header ] { declaration |
comment-run } ;
grammar-file   = [ comment-run ] grammar-header
                 { define-action | "import" STRING NL | comment-
run } ;
grammar-header = "grammar" STRING NL ;
claims-file  = [ comment-run ] claims-block { comment-run } ;
claims-block = "claims" NL
                 >>> { needs-line } claim-decl { claim-decl }
               "end" "claims" NL ;
needs-line   = "needs" ( "rooms" | "things" ) ":" name { ","
[ "and" ] name } NL
             | "needs" "verbs" ":" verb-phrase { "," verb-phrase }
NL ;
verb-phrase  = WORD { WORD } ;
claim-decl   = "claim" rest-of-line NL
                 >>> claim-predicate { needs-line } ;
claim-predicate = "the" "story" "ends" "in" WORD NL
                | "the" "story" "has" "ended" NL
                | name "was" "asked" "about" STRING NL
                | name "has" "been" WORD NL
                | condition NL ;
story-header = "story" NL
               >>> { states-line | score-line | story-on-clause |
use-line
                   | use-phrasebook-line
                   | timer-clause
                   | meta-field } ;
meta-field   = "title" ":" rest-of-line NL
             | "id" ":" rest-of-line NL
             | "story-version" ":" rest-of-line NL
             | "ifid" ":" rest-of-line NL
             | name-list-field | prose-field
             | client-config-field ;
client-config-field
             = ( "client" | "theme" | "template"
               | "default-theme" | "storage-prefix" ) ":" rest-of-
line NL
             | "themes" ":" rest-of-line NL ;
name-list-field = ( "authors" | "testers" ) ":" [ rest-of-line ]
NL
                  { >>> rest-of-line NL } ;
prose-field  = ( "prologue" | "description" ) ":" [ rest-of-line ]
```

```
NL
               { >>> rest-of-line NL } ;
use-line     = use-scoring-line | use-hunger-line | use-plain-
line ;
use-plain-line = "use" WORD [ announce-suffix ] NL ;
use-scoring-line = "use" "scoring" [ announce-suffix ] NL
               >>> { rank-line } ;
announce-suffix = "," "announce"
                  ( "all" | "collapsed" | "combined" |
"silent" ) ;
use-hunger-line = "use" "hunger" [ announce-suffix ] NL
               >>> { hunger-grows-line
                   | meter-rung-line
                   | hunger-fatal-line } ;
hunger-grows-line = "grows" NUMBER "each" "turn" NL ;
meter-rung-line = WORD "at" NUMBER [ "says" WORD ] NL ;
hunger-fatal-line = "fatal" "at" NUMBER NL ;
rank-line    = "rank" STRING "at" NUMBER [ "says" WORD ] NL ;
story-on-clause = on-clause ;
declaration  = create | start-block | opening-block
             | define-condition | define-phrase | define-phrases
             | define-trait | define-action
             | define-sequence
             | define-counter
             | define-timer
             | define-chapters
             | define-machine
             | define-fact | define-profile
             | define-mood | define-personality
             | define-temperament | define-code
             | define-honor | define-topic
             | define-asset | define-channel
             | define-family-channel
             | define-topics
             | define-manner | define-greetings
             | define-exchange | define-initiative
             | define-conversation
             | define-pronouns
             | define-phrasebook | import-phrasebook
             | override-message | override-messages
             | extend-action | remove-from-action ;
```

Message overrides

```
override-message    = "override" "message" ALIAS
                      [ "," ( STRATEGY | "verbatim" ) ] NL
                      >>> prose { "or" NL >>> prose }
                      "end" "override" NL ;
override-messages   = "override" "messages" LOCALE NL
                      >>> { ALIAS ":" NL >>> prose } ;
ALIAS               = WORD ;
```

Phrasebooks

```
define-phrasebook   = "define" "phrasebook" WORD [ "while"
condition ] NL
                      >>> { phrasebook-entry }
                      "end" "phrasebook" NL ;
phrasebook-entry    = WORD [ "," STRATEGY | "," "verbatim" ] ":"
NL
                      >>> prose { "or" NL >>> prose } ;
use-phrasebook-line = "use" "phrasebook" WORD [ "while"
condition ] NL ;
import-phrasebook   = "import" "phrasebook" STRING NL ;
```

Ownership lines

```
states-line  = "states" [ "," "reversible" ] ":" WORD { "," WORD }
NL ;
score-line   = "score" WORD "worth" NUMBER NL ;
holds-line   = "holds" NUMBER NL ;
opens-when-line = "opens" "when" "unlocked" NL ;
closes-when-line = "closes" "when" "switched" "off" NL ;
servings-line = "servings" NUMBER NL ;
vehicle-line = "movement" ":" ( "none" | "line" | "streets"
                              | "compass" | "flight"
                              | "open" ) NL
             | "carries" ":" ( "in" | "on" ) NL
             | "drive" ":" ( "state" | "control" | "player" ) NL
             | "commands" ":" WORD { "," WORD } NL
             | "stops" ":" name { "," name } NL
             | "direction" ":" ( "both" | "one-way" ) NL
             | "pace" ":" NUMBER ( "turn" | "turns" ) NL
             | "destinations" ":" name { "," name } NL
             | "aboard" ":" name { "," name } NL
             | "boarded" "through" ":" name NL
             | "stride" ":" NUMBER ( "unit" | "units" ) NL
             | sight-line
             | seen-from-afar-line
             | "sails" ":" WORD { "," WORD } NL
             | "sails" "start" ":" WORD NL
             | sailing-block ;
sailing-block = "sailing" NL
              >>> { NUMBER "to" NUMBER "degrees" ":" sailing-cells
NL
                  | "calm" ":" sailing-pace NL }
                "end" "sailing" NL ;
sailing-cells = sailing-pace
              | WORD sailing-pace { "," WORD sailing-pace } ;
sailing-pace = NUMBER ( "turn" | "turns" ) | "stopped" ;
sight-line   = "sight" ":" NUMBER NL ;
seen-from-afar-line = "seen" "from" "afar" ":" rest-of-line NL ;
expanse-line = "size" ":" NUMBER "by" NUMBER NL
             | "visibility" ":" NUMBER NL
             | "wind" ":" ( "from" ( DIRECTION | NUMBER
"degrees" )
                          | "calm" ) NL
             | "drift" ":" "1" "unit" "every" NUMBER ( "turn" |
"turns" ) NL ;
landmarks-block = "landmarks" NL
```

```
                >>> { name ":" NUMBER "," NUMBER NL }
                  "end" "landmarks" NL ;
admits-line  = "admits" ":" admits-list NL ;
admits-list  = admits-word { "," admits-word } ;
admits-word  = name | "pushing" ;
pushing-block = "pushing" NL
               >>> { ( "between" name "and" name | "from" name
"to" name ) NL }
                 "end" "pushing" NL ;
position-block = "position" NL
               >>> { "when" condition ":" name NL }
                   "otherwise" ":" name NL
                 "end" "position" NL ;
counter-line = "counter" WORD [ "starts" NUMBER ]
                         [ "between" NUMBER "and" NUMBER ] NL ;
```

On / after clauses

```
on-clause    = ( "on" | "after" ) clause-form
               [ "while" condition | "during" WORD ]
               { "," clause-modifier } NL
               >>> { statement } "end" ( "on" | "after" ) NL ;
clause-form  = head [ counterpart ]
               [ "to" WORD
               | "to" name ]
               [ "anything" "as" "the" WORD ]
             | "every" "turn" ;
head         = actor gerund { "," gerund }
             | gerund { "," gerund }
             | "hearing" ARTICLE WORD ;
actor        = "the" "player"
             | name ;
gerund       = WORD ;
counterpart  = [ "at" | "for" | "from" | "about" | "in" ] ARTICLE
name
             | ( "at" | "for" | "from" ) name ;
clause-modifier = "once"
             | ( "before" | "after" ) WORD ;
```

Start block

```
start-block  = "before" "the" "game" "starts" NL >>> { statement }
"end" "before" NL ;
```

Opening block

```
opening-block   = "when" "the" "game" "starts" NL
                  >>> { opening-routine NL } "end" "when" NL ;
opening-routine = "show-prologue"
                | "show-banner"
                | "do-first-look" ;
```

Create

```
create       = "create" name [ "," composition { ","
composition } ] NL
               >>> { create-line } ;
create-line  = "aka" alias { "," alias } NL
             | "pronouns" WORD NL
             | states-line
             | score-line
             | counter-line
             | holds-line
             | servings-line
             | opens-when-line | closes-when-line
             | vehicle-line
             | admits-line
             | position-block
             | pushing-block
             | expanse-line
             | sight-line | seen-from-afar-line
             | landmarks-block
             | composition { "," composition } NL
             | "playable" NL
             | placement NL
             | "wears" name NL
             | "carries" name NL
             | landing-line
             | room-group
             | exit-table
             | exit [ "," "one-way" ]
               [ "admits" admits-list ] [ "while" condition ] NL
             | exits-line
             | passage-block
             | DIRECTION "is" "blocked" [ "while" condition ] ":"
phrase-key NL
             | DIRECTION "is" "deadly" ":" phrase-key NL
             | phrase-override
             | room-name-block
             | "first" "time" NL >>> prose-paragraph
             | on-clause
             | timer-clause | move-clause
             | mood-line | feels-line | knows-line | thinks-line
             | spreads-line
             | resists-line
             | influence-block
             | goal-block
```

```
              | temperament-line
              | "answers" "honestly" NL
              | honor-line
              | "burdened" "by" name NL
              | "code" WORD NL
              | prose-paragraph ;
```

The character model

```
mood-line    = "mood" WORD NL ;
feels-line   = "feels" disposition [ "toward" ] name NL ;
disposition  = WORD [ WORD ] ;
knows-line   = "knows" name { "," WORD } NL ;
thinks-line  = "thinks" name "is" name { "," WORD } NL ;
spreads-line = "spreads" "nothing" NL
             | "spreads" [ topic-list ] [ "chatty" ] "to" WORD
               [ "," "except" name-ref-list ] NL ;
topic-list   = name { ( "," | "and" ) name } ;
name-ref-list = name { "," [ "and" ] name | "and" name } ;
resists-line = "resists" WORD [ "," "except" "from" name ] NL ;
influence-block = "influence" WORD { "," WORD } NL
                  >>> { influence-line }
                  "end" "influence" NL ;
influence-line = "clouds" "focus" NL
               | "makes" WORD WORD NL
               | "phrase" phrase-key "on"
                 ( "witnessed" | "resisted" | "expired" ) NL ;
define-fact  = "define" "fact" name NL
               >>> { name-ref-list NL }
               "end" "fact" NL ;
define-profile = "define" "profile" WORD NL
                 >>> { WORD WORD NL }
                 "end" "profile" NL ;
define-mood  = "define" "mood" WORD "like" WORD
               [ "," "but" WORD ] NL ;
define-personality = "define" "personality" WORD NL ;
```

The normative layer

```
temperament-line = "temperament" WORD [ "with" force-pair { "and"
force-pair } ]
                     [ "while" WORD ] NL
              | "temperament" force-pair { "and" force-pair }
                [ "while" WORD ] NL ;
force-pair   = WORD "over" WORD ;
never-line   = "never" WORD { WORD }
                [ "," "except" [ "to" "protect" ] scope-ref ] NL ;
protects-line = "protects" scope-ref NL ;
honor-line   = "honor" [ WORD ] "before" scope-ref
                [ "," "except" name-ref-list ] NL ;
scope-ref    = "anyone" | name ;
define-temperament = "define" "temperament" WORD NL
                >>> { force-pair NL }
                "end" "temperament" NL ;
define-code  = "define" "code" WORD NL
                >>> { never-line | protects-line | "answers"
"honestly" NL }
                "end" "code" NL ;
define-honor = "define" "honor" WORD NL
                >>> { WORD { WORD } NL }
                "end" "honor" NL ;
define-topic = "define" "topic" WORD WORD { WORD } "as" WORD NL ;
goal-block   = "goal" WORD "," WORD NL
                >>> { goal-line }
                "end" "goal" NL ;
goal-line    = "active" "when" condition NL
              | "seek" name [ "in" name ] NL
              | "acquire" name NL
              | "wait" "for" condition NL
              | "move" "to" name NL
              | "act" phrase-key NL
              | "say" phrase-key [ "to" name ] NL
              | "give" name "to" name NL
              | "drop" name [ "in" name ] NL
              | verb-words NL ;
```

Chapters, a `use chapters` extension

```
define-chapters = "define" "chapters" NL
                  >>> { chapter-row }
                  "end" "chapters" NL ;
chapter-row  = WORD "-" rest-of-line NL
               >>> [ prose-paragraph ]
               "begins" "when" chapter-trigger NL ;
chapter-trigger = "the" "game" "starts"
             | "the" "player" "visits" name
               "for" "the" "first" "time"
             | value-expr "expires"
             | name "becomes" WORD ;
```

Timers, places, landings

```
define-timer = "define" "timer" WORD [ "for" name ] NL
               >>> { timer-line }
               "end" "timer" NL ;
timer-line   = WORD NL [ >>> prose-paragraph ]
             | "meanwhile" [ "," "one" "chance" "in" NUMBER ] NL
               >>> { statement }
             | "interrupted" "one" "chance" "in" NUMBER NL ;
timer-clause = "when" value-expr "expires"
               [ [ "," ] "while" condition ] NL
               >>> { statement } "end" "when" NL ;
move-clause  = "when" value-expr "moves"
               [ [ "," ] "while" condition ] NL
               >>> { statement } "end" "when" NL ;
landing-line = "landing" name NL
             | "landing" "," LANDING-STRATEGY ":" name-ref-list
NL ;
LANDING-STRATEGY = "randomly" | "cycling" | "stopping" ;
phrase-override = "phrase" phrase-key
                  [ "," STRATEGY ] [ "while" condition ] ":" NL
                  variant { "or" NL variant } ;
room-name-block = "room" "name" [ "while" condition ] ":" NL
                  >>> prose-paragraph ;
room-group   = "rooms" stem NUMBER "to" NUMBER NL >>> { group-
line } ;
stem         = WORD { WORD | NUMBER } ;
group-line   = composition { "," composition } NL
             | "aka" alias { "," alias } NL
             | admits-line
             | room-name-block
             | "first" "time" NL >>> prose-paragraph
             | prose-paragraph ;
exit-table   = "exits" [ "," "one-way" ] NL >>> exit-row { exit-
row } ;
exit-row     = name ":" exit { "," exit } NL ;
exits-line   = "exits" [ "," "one-way" ] ":" exit { "," exit }
NL ;
passage-block = "passage" "to" ( name | "outside" ) [ "through"
name ] NL
              >>> "grammar" NL
                  >>> passage-pattern NL { passage-pattern NL }
                  { passage-line }
                "end" "passage" NL ;
passage-pattern = passage-elem { passage-elem } ;
```

```
passage-elem = WORD { "or" WORD } | "[" WORD { "or" WORD } "]" ;
passage-line = "while" condition NL
             | "blocked" [ "while" condition ] ":" phrase-key NL
             | "deadly" ":" phrase-key NL ;
exit         = DIRECTION "to" ( name | "outside" )
               [ "through" name ] ;
composition  = ( ARTICLE KIND-NOUN | WORD )
               [ "with" setting { "and" setting } ]
               [ "while" condition ]
             | "costs" NUMBER WORD
             | "powered" "by" ARTICLE name
               { "and" ARTICLE name }
             | "attached" "to" ARTICLE name
             | "wearable" "on" ARTICLE WORD { WORD }
             | "starts" STATE-INIT ;
STATE-INIT   = "locked" | "unlocked" | "closed" | "open"
             | "off" | "on" | "broken"
             | "lit" | "unlit" ;
KIND-NOUN    = "thing" | "room" | "door" | "person"
             | "container" | "supporter" | "region"
             | "vehicle"
             | "expanse" ;
setting      = ARTICLE name
             | WORD { WORD }
               ( NUMBER | STRING | WORD | name-list-value
               | ARTICLE name ) ;
name-list-value = "[" name { "," name } "]" ;
placement    = ( "in" | "on" ) ARTICLE name
               [ "at" NUMBER "," NUMBER ]
             | "in" ARTICLE name { "," ARTICLE name }
               "and" ARTICLE name
             | "starts" "in" name ;
DIRECTION    = "north" | "south" | "east" | "west"
             | "northeast" | "northwest" | "southeast" |
"southwest"
             | "up" | "down"
             | "in" | "out" ;
```

Define

```
define-condition = "define" "condition" WORD ":" condition NL ;
define-phrase    = "define" "phrase" phrase-key [ "," ( STRATEGY |
"verbatim" ) ]
                   [ "while" condition ] NL
                   variant { "or" NL variant } "end" "phrase" NL ;
variant          = prose-paragraph ;
define-phrases   = "define" "phrases" LOCALE NL >>> { phrase-
entry } ;
phrase-entry     = phrase-key ":" NL prose-paragraph ;
pattern          = pattern-elem { pattern-elem } ;
pattern-elem     = plain-elem | "[" plain-elem "]"
                 | "[" "the" "]" ;
plain-elem       = WORD { "or" WORD } | "the" WORD ;
STRATEGY        = "randomly" | "cycling" | "stopping" | "sticky" |
"first-time" ;
```

Traits, actions, sequences

```
define-trait   = "define" "trait" WORD NL
                 { trait-line }
                 "end" "trait" NL ;
trait-line     = "data" NL >>> { trait-field }
               | states-line
               | score-line
               | "phrases" LOCALE NL >>> { phrase-entry }
               | on-clause ;
trait-field    = field-words ":" [ "optional" ]
                 ( "entity" | "number" | "name"
                 | "one" "of" WORD { "," WORD } )
                 [ "," "starts" token ] NL ;
define-action  = "define" "action" WORD [ "while" condition ] NL
action-line* ;
action-line    = "grammar" NL >>> { pattern-line }
               | "the" WORD "must" "be" WORD NL
               | "the" WORD "takes" "the" "rest" "of" "the" "line"
NL
               | "the" WORD "is" ( "an" | "a" ) WORD NL
               | "directions" NL >>> { WORD { "or" WORD } NL }
               | must-line
               | score-line
               | "refuse" "without" WORD ":" phrase-key NL
               | "refuse" "when" condition ":" phrase-key NL
               | "otherwise" "refuse" phrase-key NL
               | topic-row
               | "phrases" LOCALE NL >>> { phrase-entry }
               | statement ;
pattern-line   = pattern [ "→" token { token } ] NL
                 { "means" WORD WORD NL } ;
```

Author alteration model

```
extend-action  = "extend" "action" WORD NL action-line* ;
remove-from-action
               = "remove" "from" "action" WORD NL
                 >>> { pattern NL } ;
define-sequence= "define" "sequence" WORD { WORD } NL
                 { sequence-step } "end" "sequence" NL ;
```

Numeric counters

```
define-counter = "define" "counter" WORD [ "starts" NUMBER ]
                           [ "between" NUMBER "and" NUMBER ] NL ;
```

State-machine depth

```
define-machine = "define" "machine" WORD { WORD } NL
                 >>> { "role" WORD "is" name NL
                     | "starts" WORD NL
                     | machine-state }
                 "end" "machine" NL ;
machine-state  = "state" WORD [ "," "terminal" ] NL
                 >>> { machine-when | machine-on } ;
machine-when   = "when" machine-trigger [ "while" condition ] ":"
WORD NL ;
machine-trigger= "event" phrase-key
               | WORD name
               | WORD
               | condition ;
machine-on     = ( "on" "enter" | "on" "exit" ) NL
                 >>> { statement } "end" "on" NL ;
```

Declared assets + custom channels

```
define-asset   = "define" ( "sound" | "image" | "music" )
                 WORD "from" STRING NL
               | "define" "sound" WORD NL
               | "define" "sound" WORD [ "from" STRING ] NL
                 >>> { sound-line } "end" "sound" NL ;
sound-line     = "heard" ":" rest-of-line NL
               | "heard" "from" "afar" ":" rest-of-line NL ;
define-family-channel
             = "define" ( "ambient" | "layer" ) WORD NL ;
define-channel = "define" "channel" WORD NL
                 >>> { "mode" ( "replace" | "append" | "event" )
NL
                     | "gated" "by" WORD NL
                     | "return" channel-return "from" WORD NL
                       [ channel-record-body ] }
                 "end" "channel" NL ;
channel-return = WORD | STRING | "phrase" WORD
               | "record" ;
channel-record-body
             = >>> { channel-record-member }
               "end" "record" NL ;
channel-record-member
             = WORD [ "list" "of" ] channel-return NL ;
define-topics  = "define" "topics" "for" name NL
                 >>> { topic-row }
                 "end" "topics" NL ;
topic-row      = "about" topic-key ":" ( statement
                                       | NL >>> { statement } ) ;
topic-key      = name
               | STRING { "," STRING } ;
define-manner  = "define" "manner" "for" name NL
                 >>> { manner-row }
                 "end" "manner" NL ;
manner-row     = "when" condition ":" NL
                 >>> { "beat" STRING NL
                     | "voice" WORD NL } ;
define-greetings = "define" "greetings" "for" name NL
                   >>> { greeting-row }
                   "end" "greetings" NL ;
greeting-row   = greeting-head ":" ( statement
                                   | NL >>> { statement } ) ;
greeting-head  = "first" "time"
               | "on" "return" [ "," absence-words ]
```

```
               | "asked" repetition-words
               | "on" "leaving" ;
absence-words  = "again" "so" "soon"
               | "after" "a" "while"
               | "after" "days" ;
repetition-words = "once" | "again" | "many" "times" ;
define-exchange = "define" "exchange" WORD "for" name
                  [ "," strength-word ] NL
                  >>> { exchange-row }
                  "end" "exchange" NL ;
exchange-row   = exchange-head ":" ( statement
                                   | NL >>> { statement } ) ;
exchange-head  = "answer" topic-key
               | "on" "silence"
               | "on" WORD ;
strength-word  = "passive" | "assertive" | "blocking" ;
define-initiative = "define" "initiative" "for" name NL
                    >>> { initiative-row }
                    "end" "initiative" NL ;
initiative-row = occasion-head [ "," "when" condition ]
                 ":" ( statement | NL >>> { statement } ) ;
occasion-head  = "on" "an" "open" "floor"
               | "on" "silence"
               | "when" "the" "subject" "changes"
               | "on" WORD ;
define-conversation = "define" "conversation" WORD "for" name
                      [ "," strength-word ] NL
                      >>> { conversation-row }
                      "end" "conversation" NL ;
conversation-row = "about" topic-key NL
                 | "opens" "when" condition NL
                 | "beat" [ "," "when" condition ] ":"
                   ( statement | NL >>> { statement } )
                 | "on" transition-word ":"
                   ( statement | NL >>> { statement } )
                 | "conclusion" ":"
                   ( statement | NL >>> { statement } ) ;
transition-word = "parting" | "resuming" | "refusing" ;
define-pronouns = "define" "pronouns" WORD NL
                  >>> { PRONOUN-CASE WORD NL }
                  "end" "pronouns" NL ;
PRONOUN-CASE    = "subject" | "object" | "possessive"
                | "possessive-pronoun" | "reflexive" ;
sequence-step  = step-anchor NL >>> { statement } ;
```

```
step-anchor    = "at" "turn" NUMBER
               | NUMBER "turns" "later"
               | "when" name "becomes" ( WORD | "under" "way" )
               | "when" name "comes" "into" "sight" ;
```

Must requirements

```
must-line    = must-subject "must" infinitive ":" phrase-key NL ;
must-subject = value-expr ;
infinitive   = "be" ( ( "a" | "an" ) WORD { WORD }
                    | "in" name
                    | "any" WORD
                    | value-expr )
             | ( "have" | "hold" | "wear" ) name
             | ( "see" | "reach" ) name ;
```

Statements

```
statement    = "refuse" phrase-key { param } NL
             | "refuse" "when" condition ":" phrase-key NL
             | must-line
             | "phrase" phrase-key { param } [ stmt-when ] NL
[ inline-prose ]
             | "emit" phrase-key
               [ "with" emit-field { "and" emit-field } ]
               [ stmt-when ] NL
             | media-statement
             | "set" value-expr "to" value-expr [ stmt-when ] NL
             | "change" "mood" "to" WORD [ stmt-when ] NL
             | "change" "feeling" "toward" name "to" disposition
               [ stmt-when ] NL
             | "change" "the" "player" "to" name [ stmt-when ] NL
             | "change" name "to" WORD [ stmt-when ] NL
             | "move" name ( "to" place | "here" | "offstage" )
               [ stmt-when ] NL
             | "make" name ( "wear" | "take" "off" ) name
               [ stmt-when ] NL
             | timer-verb value-expr
               [ "heading" ( DIRECTION | NUMBER "degrees" ) ]
               [ stmt-when ] NL
             | "send" name "to" ( name | NUMBER "," NUMBER )
               [ stmt-when ] NL
             | "kill" "the" "player" [ phrase-key ] [ stmt-when ]
NL
               [ >>> prose-paragraph ]
             | name verb-words [ stmt-when ] NL
             | "remove" name [ stmt-when ] NL
             | "award" { token } [ "," "once" ] [ stmt-when ] NL
             | ( "raise" | "lower" ) counter-ref "by" NUMBER
               [ stmt-when ] NL
             | "win" [ WORD ] [ stmt-when ] NL
             | "lose" [ WORD ] [ stmt-when ] NL
             | "then" ( "asks" | "invites" ) WORD NL
             | "deflect" "to" ( name | STRING ) NL
             | "leave" NL
             | "hold" "their" "tongue" NL
             | select-on | select-strategy | ordinal-block | each-
block ;
stmt-when    = "when" condition ;
timer-verb   = "start" | "stop" | "restart" | "reset" |
"interrupt" ;
```

```
place        = name
             | name "'s" "location" | "its" "location"
             | "a" "random" "adjacent" "room" ;
verb-words   = WORD { WORD | name } ;
phrase-key   = WORD ;
param        = "with" WORD { WORD } "=" value-expr ;
```

Emit payload + media sugar

```
emit-field   = WORD { WORD } emit-value ;
emit-value   = NUMBER | STRING
             | "[" [ emit-value { "," emit-value } ] "]"
             | "{" [ emit-field { "," emit-field } ] "}"
             | value-expr ;
media-statement
             = "play" "sound" WORD [ stmt-when ] NL
             | "play" "music" WORD [ "looping" ] [ stmt-when ] NL
             | "play" "ambient" WORD [ "in" WORD ]
               [ stmt-when ] NL
             | "stop" "music" [ stmt-when ] NL
             | "stop" "ambient" [ "in" WORD ]
               [ stmt-when ] NL
             | "show" "image" WORD [ "in" WORD ] [ stmt-when ] NL
             | "hide" "image" [ stmt-when ] NL
             | "transition" WORD [ stmt-when ] NL
             | "clear" [ stmt-when ] NL ;
select-on    = "select" "on" value-expr NL
               >>> { "when" WORD NL >>> { statement } } "end"
"select" NL ;
select-strategy = "select" STRATEGY NL
                  >>> { statement } { "or" NL >>> { statement } }
"end" "select" NL ;
ordinal-block = ORDINAL "time" NL >>> { statement } ;
ORDINAL      = "first" | "second" | "third" | "fourth" | "fifth"
             | "sixth" | "seventh" | "eighth" | "ninth" |
"tenth" ;
each-block   = "each" WORD NL
               >>> { statement } "end" "each" NL ;
```

Conditions

```
condition    = and-expr { "or" and-expr } ;
and-expr     = unary { "and" unary } ;
unary        = "not" unary
             | "(" condition ")"
             | "one" "chance" "in" NUMBER
             | "client" "has" WORD
             | ( "any" | "no" ) WORD
             | "the" "subject" "changes"
             | "asked" repetition-words
             | ( "during" | "before" | "after" ) WORD
             | WORD
             | "the" "story" "is" [ "not" ] WORD
             | value-expr predicate ;
predicate    = "is" [ "not" ] ( ( "a" | "an" ) WORD { WORD }
                              | "in" name
                              | "here"
                              | recency-word
                              | "concluded"
                              | "under" "way"
                              | "in" "sight"
                              | "close" "by"
                              | "in" "the" "distance"
                              | "on" "the" "horizon"
                              | "within" NUMBER "of"
                                ( name | NUMBER "," NUMBER )
                              | "at" name
                              | "in" "irons"
                              | "heading" ( DIRECTION
                                          | NUMBER "degrees" )
                              | word-compare NUMBER
                              | value-expr )
             | "are" [ "not" ] WORD
             | "has" [ "not" ] ( "started" | "expired" )
             | ( "has" | "holds" | "wears" ) name
             | "feels" disposition [ "toward" ] name
             | "knows" name
             | "was" "discussed"
             | "can" ( "see" | "reach" ) name
             | compare-op NUMBER ;
recency-word = "fresh" | "recent" | "stale" ;
word-compare = "at" "least" | "more" "than" | "at" "most" | "less"
"than" ;
compare-op   = ">=" | ">" | "<=" | "<" ;
```

```
counter-ref  = WORD | "its" WORD | name "'s" WORD ;
value-expr   = NUMBER | STRING
             | "the" "match"
             | "its" field-words
             | name [ "'s" field-words ]
             | WORD { WORD } ;
name         = [ ARTICLE ] WORD { WORD } [ NUMBER ] ;
ARTICLE      = "the" | "a" | "an" ;
```

Appendix B. Kinds, trait words, state words and their pairings

The closed word lists the compiler checks a story against: 9 kinds, 43 trait words and 10 state words. A word marked with a `use` line belongs to an opt-in system and needs that line in the story. Generated from the `@sharpee/chord` vocabulary (Sharpee 5.7.0 · Chord 4.1.0) by `scripts/generate-chord-appendices.cjs`.

Kinds

Kind	Needs
`container`	
`door`	
`expanse`	`use expanses`
`person`	
`region`	
`room`	
`supporter`	
`thing`	
`vehicle`	

Trait words and their settings

Trait word	Settings	Needs
`attached`	anchor: a thing	
`breakable`		
`climbable`		
`combatant`		`use combat`

Trait word	Settings	Needs
	health: a number; max-health: a number; skill: a number; base-damage: a number; armor: a number; attack-power: a number; defense: a number; experience-value: a number; hostile: a word; can-retaliate: a word; drops-inventory: a word; is-undead: a word	
`concealed`		
`costs`		
`cuttable`	tool: a thing	
`dark`		
`detachable`		
`diggable`	tool: a thing	
`drinkable`		
`edible`		
`enterable`		
`flammable`	tool: a thing	
`floats`		`use expanses`
`follower`	immediate: a word; can-move: a word; announces-movement: a word; allowed-rooms: a list; forbidden-rooms: a list	

Trait word	Settings	Needs
`fragile`		
`guard`	can-move: a word; announces-movement: a word; allowed-rooms: a list; forbidden-rooms: a list	
`hiding-spot`		
`light-source`	tool: a thing	
`listener`		
`lockable`	key: a thing	
`loud`		
`night-vision`		
`open-inventory`		
`openable`	tool: a thing	
`passive`	can-move: a word; announces-movement: a word; allowed-rooms: a list; forbidden-rooms: a list	
`patrol`	route: a list; loop: a word; wait-turns: a number; can-move: a word; announces-movement: a word; allowed-rooms: a list; forbidden-rooms: a list	
`plural`		
`powered`	source: a thing	
`proper`		
`pullable`		
`pushable`		

Trait word	Settings	Needs
`readable`		
`scenery`		
`settable`		
`smelly`		
`switchable`		
`transparent`		
`unlisted`		
`wanderer`	move-chance: a number; can-move: a word; announces-movement: a word; allowed-rooms: a list; forbidden-rooms: a list	
`weapon`	damage: a number; skill-bonus: a number; is-blessed: a word; glows-near-danger: a word	`use combat`
`wearable`		

State words

State word
`attached`
`broken`
`closed`
`lit`
`locked`
`off`
`on`

State word
open
unlocked
worn

State pairs the platform owns

A thing with the trait word already has this pair, so a story does not declare it as states.

State	Opposite	Trait word
open	closed	openable
locked	unlocked	lockable
on	off	switchable
lit	unlit	light-source
worn	unworn	wearable
whole	broken	breakable

Words after `starts`

Word	Needs the trait word
broken	breakable
closed	openable
lit	light-source
locked	lockable
off	switchable
on	switchable
open	openable
unlit	light-source
unlocked	lockable

Words after `change … to`

Word	Needs the trait word
broken	breakable
closed	openable
lit	light-source
locked	lockable
off	switchable
on	switchable
open	openable
unlit	light-source
unlocked	lockable
whole	breakable

Appendix C. The standard actions and their gerunds

Every action the platform ships: 80 actions, 77 of them with typed commands. The heading of each entry is the gerund a clause names (`on the player taking`), and the block under it is that action's entry in the standard grammar, as shipped. Generated from the standard grammar (Sharpee 5.7.0 · Chord 4.1.0) by `scripts/generate-chord-appendices.cjs`.

Verbs and the actions they reach

The first word of every typed command, and the action or actions it can reach.

Verb	Action
about	about
acquire	taking
activate	switching_on
adjust	setting
again	again
ahead	steering
alight	exiting
answer	answering
apologise	saying_sorry
apologize	saying_sorry
ascend	climbing
ask	asking, asking_for
attach	tying
attack	attacking

Verb	Action
beg	asking_for
blow	extinguishing
board	entering
break	attacking
burn	burning
buy	buying
bye	saying_goodbye
chat	talking
check	examining
chop	cutting
climb	climbing, entering, exiting
close	closing
collect	taking
come	revealing, steering
commands	help
consult	consulting
consume	eating
continue	talking
converse	talking
cover	closing
credits	about
crouch	hiding
cut	cutting
deactivate	switching_off
descend	climbing
destroy	attacking
devour	eating
dig	digging

Verb	Action
discard	dropping
disembark	exiting
display	showing
doff	taking_off
don	wearing
douse	extinguishing
down	going, pushing, steering
doze	sleeping
drag	pulling
drink	drinking
drop	dropping
duck	hiding
east	going, pushing, steering
eat	eating
embrace	kissing
empty	emptying
enter	entering
equip	wearing
examine	examining
exit	exiting, quitting
extinguish	extinguishing
extract	removing
farewell	saying_goodbye
fasten	tying
feel	touching
fight	attacking
flip	switching_off, switching_on
furl	steering

Verb	Action
g	again
get	entering, exiting, removing, taking
give	giving
go	entering, exiting, going, sleeping, talking
goodbye	saying_goodbye
grab	taking
hand	giving
hang	putting
hard	steering
head	going
hear	listening
heave	steering
help	help
hide	hiding
hit	attacking
hop	jumping
hug	kissing
hurl	throwing
i	inventory
ignite	burning
imbibe	drinking
in	going, pushing
info	about
inform	telling
inquire	asking
insert	inserting
inspect	examining

Verb	Action
inv	inventory
inventory	inventory
jump	jumping
kill	attacking
kiss	kissing
l	looking
leap	jumping
leave	exiting
let	dropping
lie	sleeping
lift	raising
light	lighting
listen	listening
load	restoring
lock	locking
look	consulting, examining, looking, looking_under, searching
lower	lowering
make	steering
move	pushing, putting
munch	eating
murder	attacking
nap	sleeping
nibble	eating
no	saying_no
north	going, pushing, steering
northeast	going, pushing, steering
northwest	going, pushing, steering
observe	examining

Verb	Action
offer	giving
open	opening
out	going, pushing
pat	touching
pay	buying
peruse	reading
pick	taking
place	putting
points	scoring
poke	touching
port	steering
power	switching_off, switching_on
present	showing
press	pushing
prod	touching
pull	pulling
purchase	buying
push	pushing
put	dropping, extinguishing, inserting, putting, wearing
q	quitting
quaff	drinking
question	asking
quit	quitting
raise	raising
read	consulting, reading
release	dropping
remove	removing, taking_off
reply	answering

Verb	Action
respond	answering
rest	sleeping
restart	restarting
restore	restoring
retrieve	taking
reveal	revealing
rotate	turning
rub	touching
rummage	searching
run	going
save	saving
say	answering, saying_goodbye
scale	climbing
score	scoring
search	searching
secure	locking
set	burning, setting, setting_course
shorten	steering
shove	pushing
show	showing
shut	closing
sip	drinking
slay	attacking
sleep	sleeping
slice	cutting
slumber	sleeping
smash	attacking
smell	smelling

Verb	Action
sniff	smelling
snuff	extinguishing
sorry	saying_sorry
south	going, pushing, steering
southeast	going, pushing, steering
southwest	going, pushing, steering
speak	talking
stand	revealing
starboard	steering
start	switching_on
steer	steering
stop	revealing, switching_off
strike	attacking
stroke	touching
study	reading
swallow	drinking
switch	switching_off, switching_on
take	removing, taking, taking_off
talk	talking
tell	talking, telling
the	going
throw	dropping, throwing
tie	tying
toss	throwing
touch	touching
travel	going
tug	pulling
turn	setting, switching_off, switching_on, turning

Verb	Action
twist	turning
uncover	opening
undo	undoing
unequip	taking_off
unfasten	untying
unhide	revealing
unlock	unlocking
unsecure	unlocking
untie	untying
unwrap	opening
up	going, pushing, steering
version	version
view	examining
wait	waiting
wake	waking
walk	going
wave	waving
wear	wearing
west	going, pushing, steering
x	examining
yank	pulling
yes	saying_yes
z	waiting

about

```
grammar
  about
  info
  credits
```

again

```
grammar
  again
  g
```

answering

```
grammar
  answer the response
  reply the response
  respond the response
  say the response
the response is a topic
```

asking

```
grammar
  ask the recipient about the topic
  question the recipient about the topic
  inquire of the recipient about the topic
the topic is a topic
```

asking_for

```
grammar
  ask the recipient for the item
  beg the recipient for the item
```

attacking

```
grammar
  attack the target
  kill the target
  fight the target
  slay the target
  murder the target
  hit the target
  strike the target
  break the target
  smash the target
  destroy the target
  attack the target with or using the weapon
  kill the target with or using the weapon
  hit the target with or using the weapon
  strike the target with or using the weapon
the weapon is an instrument
```

burning

```
grammar
  burn the target
  burn the target with or using the tool
  ignite the target
  set fire to the target
the tool is an instrument
```

buying

```
grammar
  buy the item
  purchase the item
  pay for the item
```

climbing

```
grammar
  climb the target
  climb up the target
  climb down the target
  scale the target
  ascend the target
  descend the target
```

closing

```
grammar
  close the door
  shut the door
  cover the door
```

consulting

```
grammar
  consult the source about the topic
  look up the topic in the source
  read about the topic in the source
the topic is a topic
```

cutting

```
grammar
  cut the object with or using the tool
  cut the target
  slice the target
  chop the target
the tool is an instrument
```

digging

```
grammar
  dig the location with or using the tool
  dig the target
the tool is an instrument
```

drinking

```
grammar
  drink the item
  sip the item
  quaff the item
  swallow the item
  imbibe the item
  drink from the target
  sip from the target
```

dropping

```
grammar
  drop the item
  discard the item
  release the item
  put down the item
  throw away the item
  let go of the item
```

eating

```
grammar
  eat the item
  consume the item
  devour the item
  munch the item
  nibble the item
  munch on the item
  nibble on the item
```

emptying

```
grammar
  empty the target
  empty out the target
  empty the target into or in or onto or on the destination
```

entering

```
grammar
  enter the portal
  get in the portal
  get into the portal
  climb in the portal
  climb into the portal
  go in the portal
  go into the portal
  board the vehicle
    means gesture board
  get on the vehicle
```

examining

```
grammar
  examine the target
  x the target
  inspect the target
  check the target
  view the target
  observe the target
  look at the target
  look [carefully] at the target
```

exiting

```
grammar
  go out
  exit
  get out
  leave
  climb out
  exit the container
  get out of the container
  climb out of the container
  disembark
  disembark the vehicle
  get off the vehicle
  alight
```

extinguishing

```
grammar
  extinguish the target
  put out the target
  put the target out
  blow out the target
  blow the target out
  snuff the target
  snuff out the target
  douse the target
```

giving

```
grammar
  give the item to the recipient
  give the recipient the item
  offer the item to the recipient
  hand the item to the recipient
  hand the recipient the item
```

going

```
grammar
  go the direction
  walk the direction
  run the direction
  head the direction
  travel the direction
  the direction
directions
  north or n
  south or s
  east or e
  west or w
  northeast or ne
  northwest or nw
  southeast or se
  southwest or sw
  up or u
  down or d
  in or inside
  out or outside
```

help

```
grammar
  help
  commands
```

hiding

```
grammar
  hide behind the target
    means position behind
  duck behind the target
    means position behind
  crouch behind the target
    means position behind
  hide under the target
    means position under
  duck under the target
    means position under
  crouch under the target
    means position under
  hide on the target
    means position on
  hide in the target
    means position inside
  hide inside the target
    means position inside
  duck inside the target
    means position inside
```

inserting

```
grammar
  put the item in or into or inside the container
  insert the item in or into the container
```

inventory

```
grammar
  inventory
  inv
  i
```

jumping

```
grammar
  jump
  leap
  hop
  jump up
```

kissing

```
grammar
  kiss the target
  hug the target
  embrace the target
```

lighting

```
grammar
  light the target
  light the target with or using the tool
the tool is an instrument
```

listening

```
grammar
  hear
  hear the target
  listen
  listen to the target
```

locking

```
grammar
  lock the target
  lock the target with or using the key
  secure the target
the key is an instrument
```

looking

```
grammar
  look
  l
  look [around]
```

looking_under

```
grammar
  look under the target
  look beneath the target
  look underneath the target
```

lowering

```
grammar
  lower the target
```

opening

```
grammar
  open the door
  open the container with or using the tool
  unwrap the door
  uncover the door
  open up the door
the tool is an instrument
```

pulling

```
grammar
  pull the target
  drag the target
  yank the target
  tug the target
```

pushing

```
grammar
  push the target
  press the target
  shove the target
  move the target
  push the target the direction
  shove the target the direction
  move the target the direction
directions
  north or n
  south or s
  east or e
  west or w
  northeast or ne
  northwest or nw
  southeast or se
  southwest or sw
  up or u
  down or d
  in or inside
  out or outside
```

putting

```
grammar
  put the item on or onto the supporter
  hang the item on the hook
  move the item to the destination
  place the item in or into or inside the container
  place the item on or onto the supporter
```

quitting

```
grammar
  quit
  q
  exit game
```

raising

```
grammar
  raise the target
  lift the target
```

reading

```
grammar
  read the target
  peruse the target
  study the target
```

removing

```
grammar
  take the item from the container
  get the item from the container
  take the item from the container with or using the tool
  remove the item from the container
  extract the item from the container
the tool is an instrument
```

restarting

```
grammar
  restart
```

restoring

```
grammar
  restore
  load
  load game
  restore game
```

revealing

```
grammar
  stand up
  come out
  reveal myself
  unhide
  stop hiding
```

saving

```
grammar
  save
  save game
```

saying_goodbye

```
grammar
  goodbye
  bye
  farewell
  say goodbye
  say farewell
  goodbye to the person
  say goodbye to the person
  say farewell to the person
```

saying_no

```
grammar
  no
```

saying_sorry

```
grammar
  sorry
  apologize
  apologise
```

saying_yes

```
grammar
  yes
```

scoring

```
grammar
  score
  points
```

searching

```
grammar
  search [carefully]
  search the target
  look in or inside the target
  look through the target
  rummage in or through the target
```

setting

```
grammar
  set the target to the value
  turn the target to the value
  adjust the target to the value
  set the target
the value is a topic
```

setting_course

```
grammar
  set course for the destination
  set a course for the destination
the destination is a topic
```

showing

```
grammar
  show the item to the recipient
  show the recipient the item
  display the item to the recipient
  present the item to the recipient
```

sleeping

```
grammar
  sleep
  go to sleep
  go to bed
  lie down
  nap
  doze
  rest
  slumber
```

smelling

```
grammar
  smell
  sniff
  smell the target
  sniff the target
```

steering

```
grammar
  steer the direction
  heave to
    means gesture heave-to
  port the turn
    means gesture port
  starboard the turn
    means gesture starboard
  port
    means gesture port
  starboard
    means gesture starboard
  hard to port
    means gesture hard-port
  hard to starboard
    means gesture hard-starboard
  ahead
    means gesture ahead
  come about
    means gesture come-about
  make sail
    means gesture make-sail
  shorten sail
    means gesture shorten-sail
  furl [the] sails
    means gesture furl-sails
the turn is a topic
directions
  north or n
  south or s
  east or e
  west or w
  northeast or ne
  northwest or nw
  southeast or se
  southwest or sw
  up or u
  down or d
```

switching_off

```
grammar
  turn off the device
  switch off the device
  flip off the device
  turn the device off
  deactivate the device
  stop the device
  power off the device
```

switching_on

```
grammar
  turn on the device
  switch on the device
  flip on the device
  turn the device on
  activate the device
  start the device
  power on the device
```

taking

```
grammar
  take the item
  get the item
  grab the item
  acquire the item
  collect the item
  pick up the item
  take up the item
  retrieve the item
```

taking_off

```
grammar
  remove the item
  doff the item
  unequip the item
  take off the item
  take the item off
```

talking

```
grammar
  talk to or with the target
  speak to or with the target
  chat with the target
  converse with the target
  tell me more
  continue
  go on
```

telling

```
grammar
  tell the recipient about the topic
  inform the recipient about the topic
the topic is a topic
```

throwing

```
grammar
  throw the item at the target
  throw the item
  toss the item
  hurl the item
  throw the item to the recipient
  toss the item at the target
  toss the item to the recipient
  hurl the item at the target
  hurl the item to the recipient
```

touching

```
grammar
  touch the target
  rub the target
  feel the target
  pat the target
  stroke the target
  poke the target
  prod the target
```

turning

```
grammar
  turn the target
  rotate the target
  twist the target
```

tying

```
grammar
  tie the item to the anchor
  fasten the item to the anchor
  attach the item to the anchor
  tie the item
```

undoing

```
grammar
  undo
```

unlocking

```
grammar
  unlock the door with or using the key
  unlock the target
  unsecure the target
the key is an instrument
```

untying

```
grammar
  untie the item
  untie the item from the anchor
  unfasten the item
```

version

```
grammar
  version
```

waiting

```
grammar
  wait
  z
```

waking

```
grammar
  wake
  wake up
```

waving

```
grammar
  wave
  wave hands
  wave hand
  wave the target
```

wearing

```
grammar
  wear the item
  don the item
  equip the item
  put on the item
```

Actions with no typed command of their own

The standard grammar gives these no command: `breaking`, `entering_room`, `tasting`.

Appendix D. Statements and conditions

Every statement a clause body may hold and every test a condition may make, one form to a line, in the notation of Appendix A: 38 statements, 11 conditions and 24 predicates. Generated from `chord.ebnf` (Sharpee 5.7.0 · Chord 4.1.0) by `scripts/generate-chord-appendices.cjs`.

Statements

```
"refuse" phrase-key { param } NL
"refuse" "when" condition ":" phrase-key NL
must-subject "must" infinitive ":" phrase-key NL
"phrase" phrase-key { param } [ stmt-when ] NL [ inline-prose ]
"emit" phrase-key [ "with" emit-field { "and" emit-field } ]
[ stmt-when ] NL
"play" "sound" WORD [ stmt-when ] NL
"play" "music" WORD [ "looping" ] [ stmt-when ] NL
"play" "ambient" WORD [ "in" WORD ] [ stmt-when ] NL
"stop" "music" [ stmt-when ] NL
"stop" "ambient" [ "in" WORD ] [ stmt-when ] NL
"show" "image" WORD [ "in" WORD ] [ stmt-when ] NL
"hide" "image" [ stmt-when ] NL
"transition" WORD [ stmt-when ] NL
"clear" [ stmt-when ] NL
"set" value-expr "to" value-expr [ stmt-when ] NL
"change" "mood" "to" WORD [ stmt-when ] NL
"change" "feeling" "toward" name "to" disposition [ stmt-when ] NL
"change" "the" "player" "to" name [ stmt-when ] NL
"change" name "to" WORD [ stmt-when ] NL
"move" name ( "to" place | "here" | "offstage" ) [ stmt-when ] NL
"make" name ( "wear" | "take" "off" ) name [ stmt-when ] NL
timer-verb value-expr [ "heading" ( DIRECTION | NUMBER
"degrees" ) ] [ stmt-when ] NL
"send" name "to" ( name | NUMBER "," NUMBER ) [ stmt-when ] NL
"kill" "the" "player" [ phrase-key ] [ stmt-when ] NL [ >>> prose-
paragraph ]
name verb-words [ stmt-when ] NL
"remove" name [ stmt-when ] NL
"award" { token } [ "," "once" ] [ stmt-when ] NL
( "raise" | "lower" ) counter-ref "by" NUMBER [ stmt-when ] NL
"win" [ WORD ] [ stmt-when ] NL
"lose" [ WORD ] [ stmt-when ] NL
"then" ( "asks" | "invites" ) WORD NL
"deflect" "to" ( name | STRING ) NL
"leave" NL
"hold" "their" "tongue" NL
"select" "on" value-expr NL >>> { "when" WORD NL >>>
{ statement } } "end" "select" NL
"select" STRATEGY NL >>> { statement } { "or" NL >>>
{ statement } } "end" "select" NL
ORDINAL "time" NL >>> { statement }
"each" WORD NL >>> { statement } "end" "each" NL
```

Conditions

A condition joins tests with `and` and `or`; `not` binds tightest.

```
condition = and-expr { "or" and-expr } ;
and-expr = unary { "and" unary } ;
```

Each test is one of these.

```
"not" unary
"(" condition ")"
"one" "chance" "in" NUMBER
"client" "has" WORD
( "any" | "no" ) WORD
"the" "subject" "changes"
"asked" repetition-words
( "during" | "before" | "after" ) WORD
WORD
"the" "story" "is" [ "not" ] WORD
value-expr predicate
```

Predicates

What may follow the value a test starts with.

```
"is" [ "not" ] ( "a" | "an" ) WORD { WORD }
"is" [ "not" ] "in" name
"is" [ "not" ] "here"
"is" [ "not" ] recency-word
"is" [ "not" ] "concluded"
"is" [ "not" ] "under" "way"
"is" [ "not" ] "in" "sight"
"is" [ "not" ] "close" "by"
"is" [ "not" ] "in" "the" "distance"
"is" [ "not" ] "on" "the" "horizon"
"is" [ "not" ] "within" NUMBER "of" ( name | NUMBER "," NUMBER )
"is" [ "not" ] "at" name
"is" [ "not" ] "in" "irons"
"is" [ "not" ] "heading" ( DIRECTION | NUMBER "degrees" )
"is" [ "not" ] word-compare NUMBER
"is" [ "not" ] value-expr
"are" [ "not" ] WORD
"has" [ "not" ] ( "started" | "expired" )
( "has" | "holds" | "wears" ) name
"feels" disposition [ "toward" ] name
"knows" name
"was" "discussed"
"can" ( "see" | "reach" ) name
compare-op NUMBER
```

The rules these lines name

```
stmt-when = "when" condition ;
timer-verb = "start" | "stop" | "restart" | "reset" |
"interrupt" ;
place = name | name "'s" "location" | "its" "location" | "a"
"random" "adjacent" "room" ;
verb-words = WORD { WORD | name } ;
phrase-key = WORD ;
param = "with" WORD { WORD } "=" value-expr ;
must-subject = value-expr ;
infinitive = "be" ( ( "a" | "an" ) WORD { WORD } | "in" name |
"any" WORD | value-expr ) | ( "have" | "hold" | "wear" ) name |
( "see" | "reach" ) name ;
emit-field = WORD { WORD } emit-value ;
emit-value = NUMBER | STRING | "[" [ emit-value { "," emit-
value } ] "]" | "{" [ emit-field { "," emit-field } ] "}" | value-
expr ;
disposition = WORD [ WORD ] ;
repetition-words = "once" | "again" | "many" "times" ;
recency-word = "fresh" | "recent" | "stale" ;
word-compare = "at" "least" | "more" "than" | "at" "most" | "less"
"than" ;
compare-op = ">=" | ">" | "<=" | "<" ;
counter-ref = WORD | "its" WORD | name "'s" WORD ;
value-expr = NUMBER | STRING | "the" "match" | "its" field-words |
name [ "'s" field-words ] | WORD { WORD } ;
name = [ ARTICLE ] WORD { WORD } [ NUMBER ] ;
inline-prose = >>> prose-paragraph ;
prose-paragraph = prose-line { prose-line } ;
STRATEGY = "randomly" | "cycling" | "stopping" | "sticky" |
"first-time" ;
ORDINAL = "first" | "second" | "third" | "fourth" | "fifth" |
"sixth" | "seventh" | "eighth" | "ninth" | "tenth" ;
DIRECTION = "north" | "south" | "east" | "west" | "northeast" |
"northwest" | "southeast" | "southwest" | "up" | "down" | "in" |
"out" ;
```

Appendix E. Character vocabulary

The closed word lists of the character model: 96 words in 18 lists, 5 cognitive dimensions and 8 profile presets. Generated from the `@sharpee/chord` character vocabulary (Sharpee 5.7.0 · Chord 4.1.0) by `scripts/generate-chord-appendices.cjs`.

Word lists

List	Words
Personality words	`honest`, `loyal`, `cowardly`, `paranoid`, `cruel`, `cunning`, `curious`, `stubborn`, `generous`, `vain`, `devout`, `impulsive`, `remorseful`, `untroubled`
Intensity words	`slightly`, `somewhat`, `very`, `extremely`
Moods	`calm`, `content`, `cheerful`, `nervous`, `anxious`, `panicked`, `angry`, `furious`, `sad`, `grieving`, `suspicious`, `confused`, `resigned`
Mood modifiers	`restless`, `stiller`, `darker`, `brighter`
Dispositions	`despises`, `hates`, `dislikes`, `wary of`, `neutral`, `likes`, `trusts`, `devoted to`
Threat words	`safe`, `uneasy`, `wary`, `threatened`, `cornered`, `desperate`
Confidence words, weakest first	`uncertain`, `suspects`, `believes`, `certain`
Fact sources	

List	Words
	witnessed, told, inferred, assumed, hallucinated
Resistance modes	none, reinterprets, ignores
Audiences	trusted, anyone, allied
Goal priorities	critical, high, medium, low
Influence modes	passive, active
Influence ranges	proximity, targeted, room
Forces	fear, desire, duty, honor, love
Act categories	betray a confidence, lie, harm, steal, break a promise, abandon, trespass
Obligation words	protects, answers honestly
Face acts	backs down, shows fear, admits fault, pleads, accepts insult, caught lying
Pressure bands, lightest first	clear, burdened, breaking

Cognitive dimensions

Dimension	Values
perception	accurate, filtered, augmented
belief-formation	flexible, rigid, resistant
coherence	focused, drifting, fragmented
lucidity	stable, fluctuating, episodic
self-model	intact, uncertain, fractured

Profile presets

Preset	perception	belief-formation	coherence	lucidity	self-model
braced	filtered	rigid	drifting	episodic	uncertain
clear-headed	accurate	flexible	focused	stable	intact
elsewhere	accurate	flexible	focused	episodic	fractured
fixated	accurate	resistant	focused	stable	intact
fogged	filtered	flexible	drifting	fluctuating	uncertain
loosened	filtered	flexible	drifting	fluctuating	intact
unmoored	filtered	rigid	fragmented	fluctuating	fractured
unquiet	augmented	resistant	fragmented	episodic	uncertain

Appendix F. Message aliases

Every standard message a story can replace with `override message`: 885 aliases in 78 groups, one group per action, each alias beside the text the platform prints by default. A `\n` marks a line break in the text. Generated from the output of `sharpee messages` (Sharpee 5.7.0 · Chord 4.1.0, locale en-US) by `scripts/generate-chord-appendices.cjs`.

about

Alias	Default text
about-about-compact	{verbatim:title} v{version} by {verbatim:author}
about-about-footer	Thank you for playing!
about-about-header	About {verbatim:title}
about-acknowledgments	Acknowledgments: {acknowledgments}
about-contact	Contact: {contact}
about-copyright	Copyright {copyright}
about-credits-header	Credits:
about-credits-list	{credits}
about-dedication	Dedication: {dedication}
about-description	Description: {verbatim:description}
about-engine-info	Powered by {engine} version {verbatim:engineVersion}
about-enjoy-game	We hope you enjoy playing {verbatim:title}!
about-game-info	{verbatim:title} \n Version {verbatim:version} \n By

Alias	Default text
	{verbatim:author} \n Released: {releaseDate}
about-game-info-simple	{verbatim:title} by {verbatim:author}
about-license	License: {license}
about-play-stats	Current Session: \n Time played: {playTime} \n Moves made: {sessionMoves}
about-session-info	You've been playing for {playTime} and made {sessionMoves} moves.
about-special-thanks	Special Thanks: {specialThanks}
about-success	{verbatim:title} \n Version {verbatim:version} \n By {verbatim:author} \n \n {verbatim:description}
about-technical-info	Technical Information: \n Engine: {engine} v{verbatim:engineVersion} \n Platform: Interactive Fiction
about-website	Website: {website}

again

Alias	Default text
again-nothing-to-repeat	There is nothing to repeat.

answering

Alias	Default text
answering-answered	{You} {answer}, "{verbatim:response}."

Alias	Default text
answering-no-question	No one has asked {you} anything.
answering-no-response	Answer what?
answering-not-an-answer	{capitalize the target} {verb:waits target} for an answer to the question.

asking

Alias	Default text
asking-already-told	{capitalize the target} {verb:says target}, "I already told you about that."
asking-confused	{capitalize the target} looks confused.
asking-earned-trust	{capitalize the target} {verb:says target}, "Since you've proven yourself, I'll tell you..."
asking-explains	{capitalize the target} {verb:explains target} about {verbatim:topic}.
asking-must-do-first	{capitalize the target} {verb:says target}, "There's something you need to do first."
asking-no-idea	{capitalize the target} {verb:says target}, "No idea what you're talking about."
asking-no-target	Ask whom?
asking-no-topic	Ask about what?
asking-not-actor	{You} can only ask questions of people.
asking-not-visible	{You} {can't} see {the target}.

Alias	Default text
asking-not-yet	{capitalize the target} {verb:says target}, "I can't tell you about that yet."
asking-remembers	{capitalize the target} {verb:says target}, "Ah yes, about {verbatim:topic}..."
asking-responds	{capitalize the target} {verb:tells target} {you} about {verbatim:topic}.
asking-shrugs	{capitalize the target} shrugs.
asking-too-far	{capitalize the target} {verb:is target} too far away.
asking-unknown-topic	{capitalize the target} {verb:says target}, "I don't know anything about that."

asking_for

Alias	Default text
asking-for-already-have	{You} already {have} {the item}.
asking-for-no-item	Ask {the recipient} for what?
asking-for-no-target	Ask whom?
asking-for-not-actor	{You} can only ask people for things.
asking-for-not-visible	{You} {can't} see {the recipient}.
asking-for-refused	{capitalize the recipient} {verb:doesn't recipient} seem inclined to hand over {the item}.
asking-for-self	{You} can't ask {yourself} for things.
asking-for-too-far	

Alias	Default text
	{capitalize the recipient} {verb:is recipient} too far away.

attacking

Alias	Default text
attacking-already-dead	{capitalize the target} {verb:is target} already dead.
attacking-attack-ineffective	{Your} attack has no effect on {the target}.
attacking-attack-invulnerable	{capitalize the target} cannot be damaged.
attacking-attack-requires-weapon	{You} {need} a weapon to damage {the target}.
attacking-attack-wrong-weapon-type	That weapon won't work on {the target}.
attacking-attacked	{You} {attack} {the target}.
attacking-attacked-with	{You} {attack} {the target} with {the weapon}.
attacking-broke	{You} {break} {the target}!
attacking-combat-already-dead	{capitalize the target} {verb:is target} already dead.
attacking-combat-attack-hit	{You} {hit} {the target} for {damage} damage.
attacking-combat-attack-hit-heavy	{You} {land} a solid blow on {the target}, dealing {damage} damage!
attacking-combat-attack-hit-light	{You} {graze} {the target}, doing {damage} damage.
attacking-combat-attack-killed	{You} {have} slain {the target}!

Alias	Default text
`attacking-combat-attack-knocked-out`	{capitalize the target} collapses, unconscious!
`attacking-combat-attack-missed`	{You} {swing} at {the target} but miss!
`attacking-combat-cannot-attack`	{You} {can't} attack {the target}.
`attacking-combat-defend-blocked`	{capitalize the target} blocks {your} attack!
`attacking-combat-defend-dodged`	{capitalize the target} dodges out of the way!
`attacking-combat-defend-parried`	{capitalize the target} parries {your} attack!
`attacking-combat-ended`	The battle is over.
`attacking-combat-health-badly-wounded`	{capitalize the target} {verb:is target} badly wounded.
`attacking-combat-health-dead`	{capitalize the target} {verb:is target} dead.
`attacking-combat-health-healthy`	{capitalize the target} appears uninjured.
`attacking-combat-health-near-death`	{capitalize the target} {verb:is target} barely clinging to life!
`attacking-combat-health-unconscious`	{capitalize the target} lies unconscious.
`attacking-combat-health-wounded`	{capitalize the target} {verb:has target} been wounded.
`attacking-combat-need-weapon`	{You} {need} a weapon to attack effectively.
`attacking-combat-no-target`	Attack what?
`attacking-combat-not-hostile`	{capitalize the target} {verb:isn't target} hostile.
`attacking-combat-player-died`	{You} {have} been slain!

Alias	Default text
attacking-combat-player-resurrected	{You} {feel} life return to {your} body.
attacking-combat-special-blessed-weapon	{Your} blessed weapon burns the undead!
attacking-combat-special-sword-glows	{Your} sword glows brightly!
attacking-combat-special-sword-stops-glowing	{Your} sword's glow fades.
attacking-combat-started	Combat has begun!
attacking-combat-target-unconscious	{capitalize the target} {verb:is target} already unconscious.
attacking-debris-created	Debris from {the target} litters the area.
attacking-defends	{capitalize the target} defends against {your} attack.
attacking-destroyed	{You} {destroy} {the target}!
attacking-dodges	{capitalize the target} dodges {your} attack.
attacking-flees	{capitalize the target} flees from {you}!
attacking-hit-blindly	{You} {swing} wildly, hitting nothing.
attacking-hit-target	{You} {hit} {the target}.
attacking-hit-with	{You} {hit} {the target} with {the weapon}.
attacking-indestructible	{capitalize the target} {verb:is target} far too solid to damage.
attacking-items-spilled	{capitalize the target}'s possessions spill onto the ground.
attacking-kicked	{You} {kick} {the target}.

Alias	Default text
attacking-killed-blindly	Something dies in the darkness.
attacking-killed-target	{You} {have} defeated {the target}!
attacking-need-weapon-to-damage	{capitalize the target} requires a weapon to damage.
attacking-no-fighting	Fighting won't solve this problem.
attacking-no-target	Attack what?
attacking-not-holding-weapon	{You} aren't holding {the weapon}.
attacking-not-reachable	{You} {can't} reach {the target}.
attacking-not-visible	{You} {can't} see {the target}.
attacking-passage-revealed	A hidden passage is revealed!
attacking-peaceful-solution	Violence isn't necessary here.
attacking-punched	{You} {punch} {the target}.
attacking-retaliates	{capitalize the target} fights back!
attacking-self	Violence against {yourself} isn't the answer.
attacking-shattered	{capitalize the target} shatters!
attacking-smashed	{You} {smash} {the target} to pieces!
attacking-struck	{You} {strike} {the target}!
attacking-struck-with	{You} {strike} {the target} with {the weapon}!
attacking-target-broke	{capitalize the target} breaks!
attacking-target-damaged	{capitalize the target} shows signs of damage. ({damage} damage dealt)
attacking-target-destroyed	{capitalize the target} {verb:is target} utterly destroyed!
attacking-target-shattered	{capitalize the target} shatters into pieces!
attacking-unarmed-attack	

Alias	Default text
	{You} {attack} {the target} with {your} bare hands.
attacking-unnecessary-violence	That seems unnecessarily violent.
attacking-violence-not-the-answer	Violence is not the answer.
attacking-wrong-weapon-type	{capitalize the target} can't be damaged with that type of weapon.

burning

Alias	Default text
burning-burned	{You} {set} fire to {the item}.
burning-cant-burn	Burning {the item} would achieve nothing here.
burning-needs-tool	{You} {need} {the tool} to burn {the item}.
burning-no-target	Burn what?
burning-no-tool	{You} {need} something to burn {the item} with.
burning-not-flammable	{capitalize the item} isn't something {you} can burn.
burning-tool-not-held	{You} {aren't} holding {the tool}.
burning-wrong-tool	{capitalize the tool} won't burn {the item}.

buying

Alias	Default text
buying-already-have	{You} already {have} {the target}.
buying-bought	{You} {buy} {the target}.

Alias	Default text
buying-cant-afford	{You} can't afford {the target}.
buying-no-target	Buy what?
buying-not-for-sale	{capitalize the target} {verb:isn't target} for sale.

climbing

Alias	Default text
climbing-already-there	{You're} already on {the place}.
climbing-cant-go-that-way	{You} {can't} climb {direction:direction} from here.
climbing-climb-nowhere	Climbing {the object} would get {you} nowhere.
climbing-climbed-down	{You} {climb} down.
climbing-climbed-onto	{You} {climb} onto {the target}.
climbing-climbed-up	{You} {climb} up.
climbing-need-equipment	{You}'d need climbing equipment for that.
climbing-no-target	What do {you} want to climb?
climbing-not-climbable	{You} {can't} climb {the object}.
climbing-nothing-to-climb	There's nothing to climb here.
climbing-too-dangerous	That looks too dangerous to climb.
climbing-too-high	That's too high to climb.
climbing-too-slippery	It's too slippery to climb.

closing

Alias	Default text
closing-already-closed	

Alias	Default text
	{capitalize the item} {verb:is item} already closed.
closing-cant-reach	{You} {can't} reach {the item}.
closing-closed	{You} {close} {the item}.
closing-no-target	Close what?
closing-not-closable	{capitalize the item} can't be closed.
closing-prevents-closing	{You} {can't} close {the item} while {obstacle} {verb:is obstacle} in the way.

consulting

Alias	Default text
consulting-no-target	Consult what?
consulting-no-topic	Look up what in {the source}?
consulting-not-consultable	{capitalize the source} {verb:isn't source} something {you} can look things up in.
consulting-nothing-found	{You} {find} nothing about that in {the source}.

cutting

Alias	Default text
cutting-cant-cut	Cutting {the item} would achieve nothing here.
cutting-cut	{You} {cut} {the item}.
cutting-needs-tool	{You} {need} {the tool} to cut {the item}.
cutting-no-target	Cut what?

Alias	Default text
`cutting-no-tool`	{You} {need} something to cut {the item} with.
`cutting-not-cuttable`	{capitalize the item} {verb:is item} not something {you} can cut.
`cutting-tool-not-held`	{You} {need} to be holding {the tool}.
`cutting-wrong-tool`	{capitalize the tool} won't cut {the item}.

digging

Alias	Default text
`digging-cant-dig`	Digging {the item} would achieve nothing here.
`digging-dug`	{You} {dig} {the item}.
`digging-needs-tool`	{You} {need} {the tool} to dig {the item}.
`digging-no-target`	Dig what?
`digging-no-tool`	{You} {need} something to dig {the item} with.
`digging-not-diggable`	{capitalize the item} {verb:is item} not something {you} can dig.
`digging-tool-not-held`	{You} {need} to be holding {the tool}.
`digging-wrong-tool`	{capitalize the tool} won't dig {the item}.

drinking

Alias	Default text
drinking-already-consumed	There's nothing left to drink.
drinking-bitter	{You} {drink} {the item}. It's quite bitter.
drinking-container-closed	{You} {need} to open {the item} first.
drinking-drunk	{You} {drink} {the item}.
drinking-drunk-all	{You} {drink} all of {the item}.
drinking-drunk-from	{You} {drink} from {the item}.
drinking-drunk-some	{You} {drink} some of {the item}.
drinking-empty-now	{You} {drink} the last of the {liquidType}.
drinking-from-container	{You} {drink} the {liquidType} from {the item}.
drinking-gulped	{You} {gulp} down {the item}.
drinking-healing	{You} {drink} {the item}. {You} {feel} better!
drinking-magical-effects	{You} {drink} {the item}. {You} {feel} strange...
drinking-no-item	Drink what?
drinking-not-drinkable	That's not something {you} can drink.
drinking-not-reachable	{You} {can't} reach {the item}.
drinking-not-visible	{You} {can't} see {the item}.
drinking-quaffed	{You} {quaff} {the item} heartily.
drinking-refreshing	{You} {drink} {the item}. How refreshing!
drinking-satisfying	{You} {drink} {the item}. That hits the spot.

Alias	Default text
drinking-sipped	{You} {take} a sip of {the item}.
drinking-some-remains	{You} {drink} some {liquidType}. Some remains.
drinking-still-thirsty	{You} {drink} {the item}, but {you're} still thirsty.
drinking-strong	{You} {drink} {the item}. It's strong!
drinking-sweet	{You} {drink} {the item}. It's sweet.
drinking-thirst-quenched	{You} {drink} {the item}. {Your} thirst is quenched.

dropping

Alias	Default text
dropping-dropped	Dropped.
dropping-dropped-afloat	{capitalize the item} {verb:floats item} where {verb:it item} {verb:falls item}.
dropping-dropped-breaks	{capitalize the item} {verb:hits item} the floor and {verb:breaks item}.
dropping-dropped-in	{You} {put} {the item} in {the container}.
dropping-dropped-multi	{item}: Dropped.
dropping-dropped-on	{You} {put} {the item} on {the surface}.
dropping-lost-overboard	{capitalize the item} drifts away.
dropping-no-target	Drop what?
dropping-not-held	{You} aren't holding {the item}.
dropping-nothing-to-drop	{You} aren't carrying anything.

Alias	Default text
dropping-still-worn	{You}'ll need to take off {the item} first.

eating

Alias	Default text
eating-already-consumed	There's nothing left of {the item} to eat.
eating-awful	{You} {eat} {the item}. It tastes awful!
eating-bland	{You} {eat} {the item}. It's rather bland.
eating-delicious	{You} {eat} {the item}. Delicious!
eating-devoured	{You} {devour} {the item} hungrily.
eating-eaten	{You} {eat} {the item}.
eating-eaten-all	{You} {eat} all of {the item}.
eating-eaten-portion	{You} {eat} a portion of {the item}.
eating-eaten-some	{You} {eat} some of {the item}.
eating-filling	{You} {eat} {the item}. That was filling.
eating-is-drink	{You} should drink {the item}, not eat it.
eating-munched	{You} {munch} on {the item}.
eating-nibbled	{You} {nibble} on {the item}.
eating-no-item	Eat what?
eating-not-edible	That's not something {you} can eat.
eating-not-reachable	{You} {can't} reach {the item}.
eating-not-visible	{You} {can't} see {the item}.
eating-poisonous	{You} {eat} {the item}. It tastes strange...

Alias	Default text
`eating-satisfying`	{You} {eat} {the item}. Very satisfying!
`eating-still-hungry`	{You} {eat} {the item}, but {you're} still hungry.
`eating-tasted`	{You} {taste} {the item}.
`eating-tasty`	{You} {eat} {the item}. It's quite tasty.

`emptying`

Alias	Default text
`emptying-already-empty`	{capitalize the target} {verb:is target} already empty.
`emptying-cant-empty-into`	{You} {can't} empty {the target} into {the destination}.
`emptying-container-closed`	{capitalize the target} {verb:is target} closed.
`emptying-destination-closed`	{capitalize the destination} {verb:is destination} closed.
`emptying-destination-full`	{capitalize the destination} {verb:has destination} no room for {the item}.
`emptying-emptied-here`	{You} {empty} {the target} onto the floor: {items}.
`emptying-emptied-into`	{You} {empty} {the target} into {the destination}: {items}.
`emptying-into-itself`	{You} {can't} empty {the target} into itself.
`emptying-no-target`	Empty what?
`emptying-not-a-container`	{You} {can't} empty {the target}.

entering

Alias	Default text
entering-already-inside	{You're} already in {the place}.
entering-boarded	{You} {board} {the vehicle}.
entering-cant-enter	{You} {can't} enter {the place}: {reason}.
entering-container-closed	{capitalize the container} {verb:is container} closed.
entering-entered	{You} {get} into {the place}.
entering-entered-on	{You} {get} onto {the place}.
entering-no-target	Enter what?
entering-not-enterable	{You} {can't} enter {the place}.
entering-not-here	{You} {don't} see {the place} here.
entering-occupied	{capitalize the place} {verb:is place} already occupied.
entering-too-full	{capitalize the place} {verb:is place} full (maximum {max} occupants).
entering-too-small	{capitalize the place} {verb:is place} too small for {you} to enter.
entering-vehicle-not-boardable	{You} {can't} board {the vehicle}.

examining

Alias	Default text
examining-brief-description	{verbatim:description}
examining-cant-see	{You} {can't} see {the item} here.
examining-container-closed	{capitalize the item} {verb:is item} closed.
examining-container-contents	In {the container} {you} {see} {items}.

Alias	Default text
examining-container-empty	{capitalize the item} {verb:is item} empty.
examining-container-open	{capitalize the item} {verb:is item} open.
examining-default-description	{capitalize the item} {verb:is item} just {a item}.{slot:detail}
examining-default-description-self	As good-looking as ever.
examining-description	{verbatim:description}{slot:detail}
examining-examined	{verbatim:description}{slot:detail}
examining-examined-container	{verbatim:description}{slot:detail}
examining-examined-door	{verbatim:description}{slot:detail}
examining-examined-readable	{verbatim:description}{slot:detail}
examining-examined-self	{verbatim:description}{slot:detail}
examining-examined-supporter	{verbatim:description}{slot:detail}
examining-examined-switchable	{verbatim:description}{slot:detail}
examining-examined-wall	{verbatim:description}{slot:detail}
examining-examined-wearable	{verbatim:description}{slot:detail}
examining-no-description	{You} {see} nothing special about {the item}.
examining-no-target	Examine what?
examining-not-visible	{You} {can't} see {the item} here.
examining-nothing-special	{You} {see} nothing special about {the item}.
examining-seen-from-afar	{verbatim:description}
examining-surface-contents	On {the surface} {you} {see} {items}.
examining-worn-by-other	{actor} {verb:is actor} wearing {the item}.
examining-worn-by-you	{You} {are} wearing {the item}.

exiting

Alias	Default text
exiting-already-outside	{You're} not inside anything.
exiting-cant-exit	{You} {can't} exit {the place}.
exiting-container-closed	{capitalize the container} {verb:is container} closed.
exiting-exit-blocked	The way out is blocked.
exiting-exited	{You} {get} out of {the place}.
exiting-exited-from	{You} {get} {preposition} {the place}.
exiting-must-stand-first	{You}'ll need to stand up first.
exiting-not-in-that	But {you} {aren't} in {the container}.
exiting-not-on-that	But {you} {aren't} on {the container}.
exiting-nowhere-to-go	There's nowhere to go from here.
exiting-outside-ambiguous	{You} ponder the reality of being outside...
exiting-vehicle-not-at-hatch	There's no way off from here.

extinguishing

Alias	Default text
extinguishing-extinguished	{You} {put} out {the target}.
extinguishing-extinguished-into-darkness	{You} {put} out {the target}, and darkness closes in.
extinguishing-no-target	Put out what?
extinguishing-not-light-source	{capitalize the target} isn't something {you} can put out.
extinguishing-not-lit	

Alias	Default text
	{capitalize the target} {verb:isn't target} lit.

giving

Alias	Default text
giving-accepts	{capitalize the recipient} accepts {the item}.
giving-given	{You} {give} {the item} to {the recipient}.
giving-gratefully-accepts	{capitalize the recipient} gratefully accepts {the item}.
giving-inventory-full	{capitalize the recipient} says, "I can't carry any more."
giving-no-item	Give what?
giving-no-recipient	Give it to whom?
giving-not-actor	{You} can only give things to people.
giving-not-holding	{You} aren't holding {the item}.
giving-not-interested	{capitalize the recipient} {verb:doesn't recipient} seem interested in {the item}.
giving-recipient-not-reachable	{capitalize the recipient} {verb:is recipient} too far away.
giving-recipient-not-visible	{You} {can't} see {the recipient}.
giving-refuses	{capitalize the recipient} politely declines.
giving-reluctantly-accepts	{capitalize the recipient} reluctantly takes {the item}.
giving-self	{You} already {have} {the item}!

Alias	Default text
giving-too-heavy	{capitalize the recipient} says, "That's too heavy for me."

going

Alias	Default text
going-already-there	{You're} already there.
going-arrived	{You} {arrive}.
going-arrives	{You} {enter} from the {direction:direction}.
going-cant-go	{You} {can't} go that way.
going-cant-go-through	{You} {can't} go through {obstacle}.
going-contents-list	{You} can {see} {items} here.
going-departs	{You} {leave} to the {direction:direction}.
going-destination-not-found	{You} {can't} go that way.
going-door-closed	{capitalize the door} {verb:is door} closed.
going-door-locked	{capitalize the door} {verb:is door} locked.
going-expanse-edge	{You} can go no farther that way.
going-moved	{You} {go} {direction:direction}.
going-movement-blocked	{verbatim:message}
going-need-light	It's too dark to go that way safely.
going-no-direction	{You}'ll have to say which direction to go.
going-no-exit	{You} {can't} go that way.
going-no-exit-that-way	{You} {can't} go that way.
going-no-exits	There are no obvious exits.

Alias	Default text
`going-not-in-room`	{You're} not in a place where {you} can go anywhere.
`going-nowhere-to-go`	{You}'ll have to say which compass direction to go in.
`going-room-description`	{name} `\n` {verbatim:description}
`going-sailing-edge`	Wind and sea drive {you} back.
`going-stays-by-vehicle`	{You} {stay} where {you} {are}, by the vehicle.
`going-thing-drifted`	{capitalize the thing} drifts.
`going-thing-washed-up`	{capitalize the thing} washes up here.
`going-too-dark`	It is pitch dark. {You} {be} likely to be eaten by a grue.
`going-vehicle-alongside`	{capitalize the vehicle} comes alongside {the thing}.
`going-vehicle-arrived`	{capitalize the vehicle} arrives at {the stop}.
`going-vehicle-at-end`	{capitalize the vehicle} is at the end of its line.
`going-vehicle-cannot-go`	{capitalize the vehicle} {can't} go that way.
`going-vehicle-cannot-walk`	{You} {can't} go anywhere from {the vehicle}.
`going-vehicle-departed`	{capitalize the vehicle} pulls away.
`going-vehicle-in-irons`	{capitalize the vehicle} {verb:is vehicle} in irons.
`going-vehicle-ride`	{You're} riding, not walking.
`going-vehicle-stopped`	{capitalize the vehicle} comes to a stop.
`going-vehicle-under-way`	

Alias	Default text
	{capitalize the vehicle} is under way.
`going-went`	{You} {go} {direction:direction}.

`help`

Alias	Default text
`help-first-time`	New to Interactive Fiction? Try these commands to get started: \n - LOOK to see where you are \n - INVENTORY to see what you're carrying \n - EXAMINE interesting objects \n - Go in compass directions (NORTH, SOUTH, etc.)
`help-general`	Welcome to Interactive Fiction! \n \n Basic commands: \n - LOOK (L): Examine your surroundings \n - INVENTORY (I): List what you're carrying \n - EXAMINE (X) [object]: Look at something closely \n - TAKE/DROP [object]: Pick up or put down items \n - GO [direction] or just [direction]: Move around \n \n For more help on a specific topic, type HELP [topic].
`help-help-footer`	For a complete list of commands, consult the game documentation.
`help-help-movement`	Movement commands: \n - GO NORTH/SOUTH/EAST/WEST (or just N/S/E/W) \n - UP/DOWN (U/D) \n - IN/OUT \n - ENTER [place] \n - EXIT

Alias	Default text
help-help-objects	Object commands: \n - TAKE/GET [object] \n - DROP [object] \n - EXAMINE/LOOK AT [object] \n - OPEN/CLOSE [object] \n - PUT [object] IN/ON [container] \n - WEAR/REMOVE [clothing]
help-help-special	Special commands: \n - SAVE/RESTORE: Save and load your game \n - SCORE: Check your progress \n - WAIT (Z): Let time pass \n - AGAIN (G): Repeat last command \n - QUIT: Exit the game
help-stuck-help	If you're stuck, try: \n - LOOK around carefully \n - EXAMINE everything \n - Check your INVENTORY \n - Try different verbs with objects
help-topic	Help on {verbatim:topic}:
help-unknown-topic	No help available on '{verbatim:topic}'. Type HELP for general help.

hiding

Alias	Default text
hiding-already-hidden	{You're} already hidden.
hiding-behind	{You} {slip} behind {the target}.
hiding-cant-hide-there-behind	{You} {can't} hide behind {the target}.
hiding-cant-hide-there-inside	{You} {can't} hide inside {the target}.

Alias	Default text
hiding-cant-hide-there-on	{You} {can't} hide on {the target}.
hiding-cant-hide-there-under	{You} {can't} hide under {the target}.
hiding-inside	{You} {climb} into {the target}, concealing {yourself}.
hiding-nothing-to-hide	{You} {can't} hide there.
hiding-on	{You} {crouch} on {the target}, out of sight.
hiding-under	{You} {crawl} under {the target}.

hunger

Alias	Default text
hunger-crossed	The hunger sharpens.
hunger-starved	{You} {have} starved to death.

inserting

Alias	Default text
inserting-already-there	{capitalize the item} {verb:is item} already in {the destination}.
inserting-container-closed	{capitalize the container} {verb:is container} closed.
inserting-inserted	{You} {insert} {the item} into {the container}.
inserting-no-destination	Insert {the item} into what?
inserting-no-room	There's no room in {the container}.
inserting-no-target	Insert what?
inserting-not-container	{You} {can't} insert things into {the destination}.

Alias	Default text
inserting-not-held	{You} {need} to be holding {the item} first.
inserting-not-insertable	{capitalize the item} can't be inserted into things.
inserting-wont-fit	{capitalize the item} won't fit in {the container}.

inventory

Alias	Default text
inventory-burden-heavy	{You're} carrying quite a load.
inventory-burden-light	{You're} traveling light.
inventory-burden-overloaded	{You're} weighed down with everything {you're} carrying.
inventory-carrying	{You} {be} carrying:
inventory-carrying-and-wearing	{You} {be} carrying and wearing:
inventory-carrying-count	{You} {be} carrying {holdingCount} item(s).
inventory-empty	{You} aren't carrying anything.
inventory-hands-empty	{Your} hands are empty.
inventory-holding-list	{items}
inventory-inventory-empty	{You} aren't carrying anything.
inventory-inventory-header	{You} {be} carrying:
inventory-item-list	{item}
inventory-nothing-at-all	{You} aren't carrying anything at all.
inventory-pockets-empty	{Your} pockets are empty.
inventory-wearing	{You} {be} wearing:
inventory-wearing-count	{You} {be} wearing {wearingCount} item(s).

Alias	Default text
inventory-worn-list	{items} (worn)

jumping

Alias	Default text
jumping-jumped	{You} {jump} on the spot. Nothing comes of it.

kissing

Alias	Default text
kissing-kissed-person	{capitalize the target} {verb:is target} having none of that.
kissing-kissed-thing	{You} {kiss} {the target}. Nothing comes of it.
kissing-no-target	Kiss what?

lighting

Alias	Default text
lighting-already-lit	{capitalize the target} {verb:is target} already lit.
lighting-lit	{You} {light} {the target}.
lighting-lit-in-darkness	{You} {light} {the target}, and the darkness draws back.
lighting-needs-tool	{You} {need} {the tool} to light {the item}.
lighting-no-fuel	{capitalize the target} won't light; it has nothing left to burn.
lighting-no-target	Light what?

Alias	Default text
`lighting-no-tool`	{You} {need} something to light {the item} with.
`lighting-not-light-source`	{capitalize the target} isn't something {you} can light.
`lighting-tool-not-held`	{You} {aren't} holding {the tool}.
`lighting-wrong-tool`	{capitalize the tool} won't light {the item}.

`listening`

Alias	Default text
`listening-active-devices`	{You} can {hear} {devices} operating nearby.
`listening-ambient-sounds`	{You} {hear} the usual ambient sounds.
`listening-container-sounds`	{You} {hear} faint sounds from inside {the target}.
`listening-device-off`	{capitalize the target} {verb:is target} silent.
`listening-device-running`	{capitalize the target} {verb:is target} making a soft humming sound.
`listening-liquid-sounds`	{You} {hear} liquid sloshing in {the target}.
`listening-listened-environment`	{You} {listen} carefully.
`listening-listened-to`	{You} {listen} carefully to {the target}.
`listening-no-sound`	{capitalize the target} {verb:isn't target} making any sound.

Alias	Default text
`listening-not-visible`	{You} {can't} see {the target} well enough to focus on its sounds.
`listening-silence`	{You} {hear} nothing out of the ordinary.
`listening-sounds-from-target`	{capitalize the target} {verb:is target} making a noise.
`listening-sounds-nearby`	{You} {hear} {sources}.

`locking`

Alias	Default text
`locking-already-locked`	{capitalize the item} {verb:is item} already locked.
`locking-cant-reach`	{You} {can't} reach {the item}.
`locking-key-not-held`	{You} {need} to be holding {the key}.
`locking-locked`	{You} {lock} {the item}.
`locking-locked-with`	{You} {lock} {the item} with {the key}.
`locking-no-key`	What do {you} want to lock it with?
`locking-no-target`	Lock what?
`locking-not-closed`	{You} {need} to close {the item} first.
`locking-not-lockable`	{capitalize the item} can't be locked.
`locking-wrong-key`	{capitalize the key} doesn't fit {the item}.

looking

Alias	Default text
looking-container-contents	In {the container} {you} {see} {items}.
looking-contents-list	{You} can {see} {items} here.
looking-examine-surroundings	{name} \n {verbatim:description}
looking-hidden-at	({You're} hiding {verbatim:position} {the spot}.)
looking-in-container	{You} {be} in {the container}.
looking-nothing-special	{You} {see} nothing special.
looking-on-supporter	{You} {be} on {the supporter}.
looking-room-dark	It's pitch dark, and {you} {can't} see a thing.
looking-room-description	{name} \n {verbatim:description}
looking-sighting-close-by	{capitalize the thing}, close by, to the {direction:direction}.
looking-sighting-here	{capitalize the thing}, right here.
looking-sighting-in-the-distance	{capitalize the thing}, in the distance, to the {direction:direction}.
looking-sighting-on-the-horizon	{capitalize the thing}, on the horizon, to the {direction:direction}.
looking-surface-contents	On {the surface} {you} {see} {items}.
looking-you-see	{You} can {see} {items} here.

looking_under

Alias	Default text
looking-under-found-under	Under {the target} {you} {find} {items}.
looking-under-no-target	Look under what?
looking-under-nothing-under	{You} {find} nothing under {the target}.

lowering

Alias	Default text
lowering-already-down	That's already lowered.
lowering-cant-lower-that	{You} {can't} lower {the target}.
lowering-lowered	{You} {lower} {the target}.
lowering-no-target	Lower what?

opening

Alias	Default text
opening-already-open	{capitalize the item} {verb:is item} already open.
opening-cant-reach	{You} {can't} reach {the item}.
opening-its-empty	{You} {open} {the container}, which is empty.
opening-locked	{capitalize the item} {verb:is item} locked.
opening-needs-tool	{You} {need} {the tool} to open {the item}.
opening-no-target	Open what?
opening-no-tool	

Alias	Default text
	{You} {need} something to open {the item} with.
opening-not-openable	{capitalize the item} can't be opened.
opening-opened	{You} {open} {the item}.
opening-revealing	Opening {the container} reveals {items}.
opening-tool-not-held	{You} {need} to be holding {the tool}.
opening-wrong-tool	{capitalize the tool} won't open {the item}.

pulling

Alias	Default text
pulling-cant-pull-that	{capitalize the target} {verb:is target} not something {you} can pull.
pulling-detached	{You} {pull} {the target} free of {the anchor}.
pulling-no-target	Pull what?
pulling-not-reachable	{You} {can't} reach {the target}.
pulling-not-visible	{You} {can't} see {the target}.
pulling-pulled	{You} {pull} {the target}.
pulling-wont-budge	{capitalize the target} won't budge.
pulling-worn	{You} {can't} pull {the target} while wearing it.

pushing

Alias	Default text
pushing-button-clicks	{You} {press} {the target}. Click!
pushing-button-pushed	{You} {push} {the target}.
pushing-cannot-push-that-way	{You} {can't} push {the target} that way.
pushing-carrying-it	{You} {can't} push {the target} while carrying it.
pushing-fixed-in-place	{capitalize the target} {verb:is target} fixed in place.
pushing-no-target	Push what?
pushing-not-reachable	{You} {can't} reach {the target}.
pushing-not-visible	{You} {can't} see {the target}.
pushing-pushed-direction	{You} {push} {the target} {direction:direction}.
pushing-pushed-nudged	{You} {give} {the target} a push, but it doesn't move far.
pushing-pushing-does-nothing	Pushing {the target} has no effect.
pushing-switch-toggled	{You} {push} {the target}, toggling it {newState}.
pushing-too-heavy	{capitalize the target} {verb:is target} far too heavy to push (weighs {weight}kg).
pushing-wearing-it	{You} {can't} push {the target} while wearing it.
pushing-wont-budge	{capitalize the target} won't budge.

putting

Alias	Default text
putting-already-there	{capitalize the item} {verb:is item} already {relation} {the destination}.
putting-cant-put-in-itself	{You} {can't} put {the item} inside itself.
putting-cant-put-on-itself	{You} {can't} put {the item} on itself.
putting-container-closed	{capitalize the container} {verb:is container} closed.
putting-no-destination	Where do {you} want to put {the item}?
putting-no-room	There's no room in {the container}.
putting-no-space	There's no space on {the surface}.
putting-no-target	Put what?
putting-not-container	{You} {can't} put things in {the destination}.
putting-not-held	{You} {need} to be holding {the item} first.
putting-not-surface	{You} {can't} put things on {the destination}.
putting-put-in	{You} {put} {the item} in {the container}.
putting-put-on	{You} {put} {the item} on {the surface}.

quitting

Alias	Default text
quitting-achievements-earned	{You} earned {count} achievements during {your} play!

Alias	Default text
quitting-final-score	{Your} final score was {finalScore} out of {maxScore}.
quitting-final-stats	Final Statistics: \n Score: {finalScore}/{maxScore} \n Moves: {moves} \n Time played: {playTime}
quitting-quit-and-saved	Game saved. \n \n Thanks for playing! \n \n Final score: {finalScore} out of {maxScore} \n Moves: {moves}
quitting-quit-cancelled	Quit cancelled.
quitting-quit-confirm-query	Are {you} sure {you} want to quit?
quitting-quit-confirmed	Thanks for playing! \n \n Final score: {finalScore} out of {maxScore} \n Moves: {moves}
quitting-quit-save-query	Would {you} like to save before quitting?
quitting-quit-unsaved-query	{You} {have} unsaved progress. What would {you} like to do?

raising

Alias	Default text
raising-already-up	That's already raised.
raising-cant-raise-that	{You} {can't} raise {the target}.
raising-no-target	Raise what?
raising-raised	{You} {raise} {the target}.

reading

Alias	Default text
reading-cannot-read-now	{verbatim:reason}
reading-not-readable	There's nothing written on {the item}.
reading-read-book	{capitalize the item} reads: \n {verbatim:text}
reading-read-book-page	{capitalize the item} (page {currentPage} of {totalPages}): \n {verbatim:text}
reading-read-inscription	{capitalize the item} reads: \n {verbatim:text}
reading-read-sign	{capitalize the item} says: \n {verbatim:text}
reading-read-text	{capitalize the item} reads: \n {verbatim:text}
reading-what-to-read	What do you want to read?

removing

Alias	Default text
removing-already-have	{You} already {have} {the item}.
removing-attached	{capitalize the item} {verb:is item} fixed to {the anchor}.
removing-cant-reach	{You} {can't} reach {the item}.
removing-container-closed	{capitalize the container} {verb:is container} closed.
removing-no-source	Remove {the item} from what?
removing-no-target	Remove what?
removing-not-in-container	

Alias	Default text
	{capitalize the item} isn't in {the container}.
`removing-not-on-surface`	{capitalize the item} isn't on {the surface}.
`removing-removed-from`	{You} {take} {the item} from {the container}.
`removing-removed-from-surface`	{You} {take} {the item} from {the surface}.

`restarting`

Alias	Default text
`restarting-game-restarting`	The story restarts.
`restarting-new-game`	A new story begins.
`restarting-restart-confirm`	Are you sure you want to restart? All unsaved progress will be lost.
`restarting-restart-requested`	Restarting the story...
`restarting-restart-unsaved`	You have unsaved progress. Restart anyway?
`restarting-starting-over`	Starting over from the beginning.

`restoring`

Alias	Default text
`restoring-available-saves`	Available saves: {saves}
`restoring-choose-save`	Which save would {you} like to restore?
`restoring-confirm-restore`	Restore game from '{verbatim:saveName}'? Current progress will be lost.

Alias	Default text
`restoring-corrupt-save`	The save file '{verbatim:saveName}' appears to be corrupted.
`restoring-game-loaded`	Game loaded from '{verbatim:saveName}'.
`restoring-game-restored`	Game restored.
`restoring-import-save`	Import a save file to restore.
`restoring-incompatible-save`	This save file is from a different version and cannot be loaded.
`restoring-no-saves`	No saved games found.
`restoring-no-saves-available`	No saved games available.
`restoring-quick-restore`	Quick restore completed.
`restoring-restore-details`	Restored: {verbatim:saveName} `\n` Score: {score} `\n` Moves: {moves}
`restoring-restore-failed`	Failed to restore game.
`restoring-restore-not-allowed`	{You} cannot restore a game at this time.
`restoring-restore-successful`	{Your} saved game has been restored successfully.
`restoring-resuming-game`	Resuming {your} adventure...
`restoring-save-imported`	Save file imported successfully.
`restoring-save-not-found`	No save named '{verbatim:saveName}' was found.
`restoring-unsaved-progress`	{You} {have} unsaved progress. Restore anyway?
`restoring-welcome-back`	Welcome back! Game restored from {verbatim:saveName}.

revealing

Alias	Default text
`revealing-not-hidden`	{You're} not hiding.
`revealing-revealed`	{You} {come} out of hiding.

saving

Alias	Default text
`saving-auto-save`	Auto-saving game...
`saving-confirm-overwrite`	A save named '{verbatim:saveName}' already exists. Overwrite it?
`saving-game-saved`	Game saved.
`saving-game-saved-as`	Game saved as '{verbatim:saveName}'.
`saving-invalid-save-name`	'{verbatim:saveName}' is not a valid save name.
`saving-no-save-slots`	No save slots available.
`saving-overwrite-save`	Previous save '{verbatim:saveName}' has been overwritten.
`saving-quick-save`	Quick save completed.
`saving-save-details`	Saved: {verbatim:saveName} `\n` Score: {score} `\n` Moves: {moves}
`saving-save-exported`	Save file exported successfully.
`saving-save-failed`	Failed to save game.
`saving-save-in-progress`	Another save is already in progress.
`saving-save-not-allowed`	{You} cannot save the game at this time.
`saving-save-reminder`	

Alias	Default text
	Don't forget to save {your} game regularly!
`saving-save-slot`	Game saved to slot {verbatim:saveName}.
`saving-save-successful`	{Your} game has been saved successfully.
`saving-saved-locally`	Game saved to local storage.
`saving-saved-to-cloud`	Game saved to cloud storage.

saying_goodbye

Alias	Default text
`saying-goodbye-not-talking`	{You} {aren't} talking to anyone.
`saying-goodbye-not-talking-to`	{You} {aren't} talking to {the target}.
`saying-goodbye-said-goodbye`	{You} {say} goodbye to {the target}.

saying_no

Alias	Default text
`saying-no-nobody-asked`	Nobody asked {you} anything.

saying_sorry

Alias	Default text
`saying-sorry-nothing-to-apologize-for`	There is nothing to apologize for.

saying_yes

Alias	Default text
saying-yes-nobody-asked	Nobody asked {you} anything.

scoring

Alias	Default text
scoring-no-scoring	This isn't that kind of game.
scoring-perfect-score	{You} {have} achieved a perfect score of {maxScore} points!
scoring-promotion	{You} {have} risen to the rank of {rank}.
scoring-score-display	{You} {have} scored {score} out of a possible {maxScore}.
scoring-score-simple	{Your} score is {score} points.
scoring-score-with-rank	{You} {have} scored {score} out of {maxScore}, earning {you} the rank of {rank}.

searching

Alias	Default text
searching-container-closed	{capitalize the target} {verb:is target} closed.
searching-container-contents	In {the target} {you} {see}: {items}.
searching-empty-container	{capitalize the target} {verb:is target} empty.
searching-found-concealed-here	Hidden here, {you} {discover}: {items}.

Alias	Default text
searching-found-concealed-in-container	Hidden inside {the target}, {you} {discover}: {items}.
searching-found-concealed-on-supporter	Hidden on {the target}, {you} {discover}: {items}.
searching-found-items	{You} {discover}: {items}.
searching-looked-through	{You} {look} through {the target}.
searching-looked-through-nothing	{You} {see} nothing through {the target}.
searching-not-reachable	{You} {can't} reach {the target} to search it.
searching-not-visible	{You} {can't} see {the target} to search it.
searching-nothing-special	{You} {find} nothing of interest.
searching-searched-location	{You} {search} around carefully.
searching-searched-object	{You} {search} {the target} thoroughly.
searching-supporter-contents	On {the target} {you} {see}: {items}.

setting

Alias	Default text
setting-cant-set-that	{capitalize the target} isn't something {you} can set.
setting-no-target	Set what?
setting-no-value	Set {the target} to what?
setting-not-a-setting	{capitalize the target} goes to {values}.
setting-set	{You} {set} {the target} to {verbatim:value}.

setting_course

Alias	Default text
setting-course-already-there	{capitalize the vehicle} {verb:is vehicle} already at {the destination}.
setting-course-coordinates-outside-expanse	{verbatim:destination} lies beyond the edge.
setting-course-course-set	{You} {set} course for {the destination}.
setting-course-course-set-cell	{You} {set} course for {verbatim:x}, {verbatim:y}.
setting-course-no-destination	Set course for where?
setting-course-not-a-destination	{capitalize the vehicle} {verb:does vehicle} not go to {verbatim:destination}.
setting-course-not-in-vehicle	{You're} not aboard anything that takes a course.
setting-course-vehicle-no-such-command	{capitalize the vehicle} takes no course.
setting-course-vehicle-under-way	{capitalize the vehicle} {verb:is vehicle} already under way.

showing

Alias	Default text
showing-no-item	Show what?
showing-no-viewer	Show it to whom?
showing-not-actor	{You} can only show things to people.
showing-not-carrying	{You} aren't carrying {the item}.
showing-self	{You} {examine} {the item} closely.

Alias	Default text
showing-shown	{You} {show} {the item} to {the viewer}.
showing-viewer-examines	{capitalize the viewer} examines {the item} carefully.
showing-viewer-impressed	{capitalize the viewer} looks impressed.
showing-viewer-nods	{capitalize the viewer} nods.
showing-viewer-not-visible	{You} {can't} see {the viewer}.
showing-viewer-recognizes	{capitalize the viewer} recognizes {the item}!
showing-viewer-too-far	{capitalize the viewer} {verb:is viewer} too far away to see clearly.
showing-viewer-unimpressed	{capitalize the viewer} seems unimpressed.
showing-wearing-shown	{You} {show} {the viewer} that {you're} wearing {the item}.

sleeping

Alias	Default text
sleeping-already-well-rested	{You're} already well-rested and don't feel tired.
sleeping-brief-nap	{You} {take} a brief nap.
sleeping-cant-sleep-here	{You} {can't} sleep in {location}.
sleeping-deep-sleep	{You} {fall} into a deep, restful sleep.
sleeping-disturbed-sleep	{Your} sleep is disturbed.
sleeping-dozed-off	{You} {doze} off for a bit.
sleeping-fell-asleep	{You} {fall} into a deep sleep.
sleeping-nightmares	{You} {have} unsettling dreams.

Alias	Default text
sleeping-not-tired	{You} aren't tired.
sleeping-peaceful-sleep	{You} {enjoy} a peaceful sleep.
sleeping-slept	{You} {sleep} for a while.
sleeping-slept-fitfully	{You} {sleep} fitfully.
sleeping-too-dangerous-to-sleep	It's too dangerous to sleep in {location}.
sleeping-woke-refreshed	{You} {wake} feeling refreshed.

smelling

Alias	Default text
smelling-burning-scent	{capitalize the target} gives off a smoky smell.
smelling-container-food-scent	{You} {smell} food inside {the target}.
smelling-declared-scent	{capitalize the target} {verb:has target} a distinct smell.
smelling-drink-scent	{capitalize the target} {verb:has target} a pleasant aroma.
smelling-food-nearby	{You} {smell} food nearby.
smelling-food-scent	{capitalize the target} smells delicious.
smelling-fresh-scent	{capitalize the target} smells fresh and clean.
smelling-musty-scent	{capitalize the target} smells a bit musty.
smelling-no-particular-scent	{capitalize the target} {verb:has target} no particular smell.
smelling-no-scent	{You} {don't} smell anything unusual.

Alias	Default text
`smelling-not-visible`	{You} {can't} see {the target} to smell it.
`smelling-room-scents`	The air carries various scents.
`smelling-scents-nearby`	{You} {smell} {sources}.
`smelling-smelled`	{You} {smell} {the target}.
`smelling-smelled-environment`	{You} {sniff} the air.
`smelling-smoke-detected`	{You} {detect} a faint smell of smoke.
`smelling-too-far`	{capitalize the target} {verb:is target} too far away to smell.

steering

Alias	Default text
`steering-ahead`	{capitalize the vehicle} gathers way on {verbatim:bearing} degrees.
`steering-came-about`	{You} {come} about, and {the vehicle} settles on {verbatim:bearing} degrees.
`steering-cannot-steer`	{You} {can't} steer {the vehicle} that way.
`steering-hove-to`	{You} {heave} to, and {the vehicle} loses way.
`steering-naval-cannot-come-about`	{capitalize the vehicle} {verb:lies vehicle} dead to the wind, with no other side to come about to.
`steering-naval-how-far`	How many degrees?
`steering-naval-no-wind`	There is no wind to come about through.
`steering-not-in-vehicle`	

Alias	Default text
	{You're} not aboard anything {you} can steer.
steering-not-under-way	{capitalize the vehicle} {verb:is vehicle} not under way.
steering-sail-made	{You} {make} sail.
steering-sail-shortened	{You} {shorten} sail.
steering-sails-already-furled	The sails are already furled.
steering-sails-at-full	All sail is set.
steering-sails-furled	{You} {furl} the sails.
steering-steered	{You} {steer} {the vehicle} {direction:direction}.
steering-turned	{You} {bring} {the vehicle} to {verbatim:side}, to {verbatim:bearing} degrees.
steering-vehicle-no-such-command	{You} {can't} steer {the vehicle}.

switching_off

Alias	Default text
switching-off-already-off	{capitalize the target} {verb:is target} already off.
switching-off-device-stops	{capitalize the target} powers down with a soft whir.
switching-off-door-closes	{capitalize the target} switches off and closes.
switching-off-light-off	{You} {switch} off {the target}, plunging the area into darkness.
switching-off-light-off-still-lit	{You} {switch} off {the target}.

Alias	Default text
switching-off-no-target	Switch off what?
switching-off-not-reachable	{You} {can't} reach {the target}.
switching-off-not-switchable	{capitalize the target} isn't something {you} can switch off.
switching-off-not-visible	{You} {can't} see {the target}.
switching-off-silence-falls	{You} {switch} off {the target}. Silence falls.
switching-off-switched-off	{You} {switch} off {the target}.
switching-off-was-temporary	{capitalize the target} switches off (it had {remainingTime} seconds left).
switching-off-with-sound	{You} {switch} off {the target}. {sound}

switching_on

Alias	Default text
switching-on-already-on	{capitalize the target} {verb:is target} already on.
switching-on-device-humming	{capitalize the target} hums to life.
switching-on-door-opens	{capitalize the target} switches on and opens.
switching-on-illuminates-darkness	{capitalize the target} switches on, banishing the darkness.
switching-on-light-on	{You} {switch} on {the target}.
switching-on-no-power	{capitalize the target} {verb:has target} no power source.
switching-on-no-target	Switch on what?
switching-on-not-reachable	{You} {can't} reach {the target}.
switching-on-not-switchable	

Alias	Default text
	{capitalize the target} isn't something {you} can switch on.
switching-on-not-visible	{You} {can't} see {the target}.
switching-on-switched-on	{You} {switch} on {the target}.
switching-on-temporary-activation	{capitalize the target} switches on temporarily.
switching-on-with-sound	{You} {switch} on {the target}. {sound}

taking

Alias	Default text
taking-already-have	{You} already {have} {the item}.
taking-attached	{capitalize the item} {verb:is item} fixed to {the anchor}.
taking-cannot-take	{You} {can't} take {the item}.
taking-cant-take-room	{You} {can't} take {the item}.
taking-cant-take-self	{You} {can't} take {yourself}.
taking-container-full	{You're} carrying too much already.
taking-fixed-in-place	{capitalize the item} {verb:is item} fixed in place.
taking-no-target	Take what?
taking-nothing-to-take	{You} {take} in everything {you} {see} and {enjoy} the moment.
taking-retrieved	{You} {haul} {the item} aboard.
taking-taken	Taken.
taking-taken-from	{You} {take} {the item} from {the container}.
taking-taken-multi	{item}: Taken.
taking-too-heavy	

Alias	Default text
	{Your} load is too heavy. {You} will have to leave something behind.

taking_off

Alias	Default text
taking-off-cant-remove	{You} {can't} take off {the item}.
taking-off-no-target	Take off what?
taking-off-not-wearing	{You} aren't wearing {the item}.
taking-off-prevents-removal	{You}'ll need to take off {the blocking} first.
taking-off-removed	{You} {take} off {the item}.

talking

Alias	Default text
talking-acknowledges	{capitalize the target} acknowledges {you}.
talking-casual-greeting	{capitalize the target} {verb:says target}, "Hey!"
talking-first-meeting	{You} {introduce} {yourself} to {the target}.
talking-formal-greeting	{capitalize the target} {verb:says target}, "Good day to you."
talking-friendly-greeting	{capitalize the target} smiles in recognition.
talking-greets-again	{capitalize the target} {verb:says target}, "Hello again."
talking-greets-back	{capitalize the target} {verb:says target}, "Hello there!"

Alias	Default text
`talking-has-topics`	{capitalize the target} seems willing to discuss various topics.
`talking-no-response`	{capitalize the target} doesn't respond.
`talking-no-target`	Talk to whom?
`talking-not-actor`	{You} can only talk to people.
`talking-not-available`	{capitalize the target} doesn't want to talk right now.
`talking-not-visible`	{You} {can't} see {the target}.
`talking-nothing-to-say`	{capitalize the target} {verb:has target} nothing particular to say.
`talking-remembers-you`	{capitalize the target} {verb:says target}, "Ah, it's you again."
`talking-self`	Talking to {yourself} is a sign of madness.
`talking-talked`	{You} {greet} {the target}.
`talking-too-far`	{capitalize the target} {verb:is target} too far away for conversation.

`telling`

Alias	Default text
`telling-already-knew`	{capitalize the target} {verb:says target}, "Yes, I'm aware of that."
`telling-bored`	{capitalize the target} looks bored.
`telling-dismissive`	{capitalize the target} {verb:says target}, "So what?"
`telling-grateful`	{capitalize the target} {verb:says target}, "Thank you for telling me!"

Alias	Default text
telling-ignores	{capitalize the target} ignores what {you're} saying.
telling-informed	{You} {inform} {the target} about {verbatim:topic}.
telling-interested	{capitalize the target} listens with interest.
telling-no-target	Tell whom?
telling-no-topic	Tell them about what?
telling-not-actor	{You} can only tell things to people.
telling-not-interested	{capitalize the target} doesn't seem interested.
telling-not-visible	{You} {can't} see {the target}.
telling-told	{You} {tell} {the target} about {verbatim:topic}.
telling-too-far	{capitalize the target} {verb:is target} too far away.
telling-very-interested	{capitalize the target} {verb:says target}, "Really? Tell me more!"

throwing

Alias	Default text
throwing-bounces-off	{capitalize the item} bounces off {the target}.
throwing-breaks-against	{capitalize the item} smashes against {the target}!
throwing-breaks-on-impact	{capitalize the item} shatters on impact!
throwing-cant-throw-that-way	{You} {can't} throw {the item} that way.

Alias	Default text
`throwing-fragile-breaks`	The fragile {item} breaks into pieces.
`throwing-hits-target`	{You} {throw} {the item} at {the target}. It hits!
`throwing-lands-in`	{capitalize the item} lands in {the target}.
`throwing-lands-on`	{capitalize the item} lands on {the target}.
`throwing-lost-overboard`	{capitalize the item} drifts away.
`throwing-misses-target`	{You} {throw} {the item} at {the target}, but miss.
`throwing-no-exit`	There's no exit {direction:direction}.
`throwing-no-item`	Throw what?
`throwing-not-holding`	{You} aren't holding {the item}.
`throwing-sails-through`	{capitalize the item} sails through the exit to the {direction:direction}.
`throwing-self`	{You} {can't} throw things at {yourself}.
`throwing-target-angry`	{capitalize the target} doesn't appreciate being hit with {the item}.
`throwing-target-catches`	{capitalize the target} catches {the item}!
`throwing-target-ducks`	{capitalize the target} ducks out of the way.
`throwing-target-not-here`	{capitalize the target} isn't here.
`throwing-target-not-visible`	{You} {can't} see {the target}.
`throwing-thrown`	{You} {throw} {the item}.
`throwing-thrown-afloat`	

Alias	Default text
	{capitalize the item} {verb:floats item} where {verb:it item} {verb:falls item}.
throwing-thrown-at	{You} {throw} {the item} at {the target}.
throwing-thrown-direction	{You} {throw} {the item} {direction:direction}.
throwing-thrown-down	{You} {toss} {the item} to the ground.
throwing-too-heavy	{capitalize the item} {verb:is item} too heavy to throw far (weighs {weight}kg).

touching

Alias	Default text
touching-device-vibrating	{capitalize the target} {verb:is target} vibrating slightly.
touching-feels-hard	{capitalize the target} feels hard and solid.
touching-feels-hot	{capitalize the target} {verb:is target} hot! {You} {pull} {your} hand back quickly.
touching-feels-normal	{capitalize the target} feels as {you}'d expect.
touching-feels-smooth	{capitalize the target} feels smooth.
touching-feels-soft	{capitalize the target} feels soft.
touching-feels-warm	{capitalize the target} feels warm to the touch.
touching-feels-wet	{capitalize the target} feels damp.
touching-immovable-object	

Alias	Default text
	{capitalize the target} {verb:is target} solid and immovable.
`touching-liquid-container`	{You} {feel} liquid sloshing inside {the target}.
`touching-no-target`	Touch what?
`touching-not-reachable`	{You} {can't} reach {the target}.
`touching-not-visible`	{You} {can't} see {the target} to touch it.
`touching-patted`	{You} {pat} {the target}.
`touching-poked`	{You} {poke} {the target}.
`touching-prodded`	{You} {prod} {the target}.
`touching-stroked`	{You} {stroke} {the target}.
`touching-touched`	{You} {touch} {the target}.
`touching-touched-gently`	{You} gently {touch} {the target}.

`turning`

Alias	Default text
`turning-cant-turn-that`	{capitalize the target} isn't something {you} can turn.
`turning-crank-turned`	{You} {crank} {the target}.
`turning-dial-adjusted`	{You} {adjust} {the target} {verbatim:direction}.
`turning-dial-set`	{You} {turn} {the target} to {setting}.
`turning-dial-turned`	{You} {turn} {the target}.
`turning-flow-changes`	{You} {turn} {the target}, adjusting the flow.
`turning-key-needs-lock`	{You} {need} to put {the target} in a lock first.

Alias	Default text
`turning-key-turned`	{You} {turn} {the target} in the lock.
`turning-knob-clicks`	{You} {turn} {the target} with a click.
`turning-knob-toggled`	{You} {turn} {the target}, switching it {newState}.
`turning-knob-turned`	{You} {turn} {the target}.
`turning-mechanism-activated`	As {you} {turn} {the target}, {you} {hear} machinery activate!
`turning-mechanism-grinds`	{You} {turn} {the target}. Gears grind and machinery moves.
`turning-no-target`	Turn what?
`turning-not-reachable`	{You} {can't} reach {the target}.
`turning-not-visible`	{You} {can't} see {the target}.
`turning-nothing-happens`	{You} {turn} {the target}, but nothing happens.
`turning-requires-more-turns`	{You} {turn} {the target}. It seems to need more turning.
`turning-rotated`	{You} {rotate} {the target}.
`turning-spun`	{You} {spin} {the target}.
`turning-turned`	{You} {turn} {the target}.
`turning-valve-closed`	{You} {turn} {the target}, closing the valve.
`turning-valve-opened`	{You} {turn} {the target}, opening the valve.
`turning-wearing-it`	{You} {can't} turn {the target} while wearing it.
`turning-wheel-turned`	{You} {turn} {the target}.

tying

Alias	Default text
tying-no-anchor	What do {you} want to tie {the item} to?
tying-no-target	Tie what?
tying-not-to-people	{You} can't tie things to people.
tying-self	{You} can't tie {the item} to itself.
tying-tied	{You} {tie} {the item} to {the anchor}.

undoing

Alias	Default text
undoing-nothing-to-undo	Nothing to undo.
undoing-undo-failed	Undo failed.
undoing-undo-success	Previous turn undone.
undoing-undo-to-turn	Undone. (Now at turn {turn})

unlocking

Alias	Default text
unlocking-already-unlocked	{capitalize the item} {verb:is item} already unlocked.
unlocking-cant-reach	{You} {can't} reach {the item}.
unlocking-key-not-held	{You} {need} to be holding {the key}.
unlocking-no-key	What do {you} want to unlock it with?
unlocking-no-target	Unlock what?
unlocking-not-lockable	{capitalize the item} can't be unlocked.
unlocking-still-locked	

Alias	Default text
	{capitalize the item} {verb:is item} locked.
`unlocking-unlocked`	{You} {unlock} {the item}.
`unlocking-unlocked-and-opened`	{You} {unlock} {the item}, and it swings open.
`unlocking-unlocked-with`	{You} {unlock} {the item} with {the key}.
`unlocking-unlocked-with-and-opened`	{You} {unlock} {the item} with {the key}, and it swings open.
`unlocking-wrong-key`	{capitalize the key} doesn't fit {the item}.

untying

Alias	Default text
`untying-no-target`	Untie what?
`untying-not-tied`	{capitalize the item} {verb:isn't item} tied to anything.
`untying-untied`	{You} {untie} {the item}.

version

Alias	Default text
`version-version-compact`	{verbatim:storyTitle} v{verbatim:storyVersion} (Sharpee v{verbatim:engineVersion})
`version-version-full`	{verbatim:storyTitle} v{verbatim:storyVersion} `\n` Sharpee Engine

Alias	Default text
	v{verbatim:engineVersion} \n Built: {buildDate}
version-version-no-date	{verbatim:storyTitle} v{verbatim:storyVersion} \n Sharpee Engine v{verbatim:engineVersion}

waiting

Alias	Default text
waiting-grows-restless	{You} {grow} restless from waiting.
waiting-nothing-happens	{You} {wait}. Nothing happens.
waiting-patience-rewarded	{Your} patience is rewarded.
waiting-something-approaches	As {you} {wait}, {you} {hear} something approaching.
waiting-time-passes	Time passes...
waiting-time-runs-out	{You}'ve waited too long!
waiting-waited	Time passes.
waiting-waited-anxiously	{You} {wait} anxiously.
waiting-waited-briefly	{You} {wait} for a moment.
waiting-waited-for-event	{You} {wait} for something to happen.
waiting-waited-in-vehicle	{You} {wait} in {the vehicle}.
waiting-waited-patiently	{You} {wait} patiently.

waking

Alias	Default text
waking-already-awake	{You're} already awake.

waving

Alias	Default text
waving-not-held	{You} {aren't} holding {the target}.
waving-waved	{You} {wave} {the target} about. Nothing happens.
waving-waved-hands	{You} {wave}. Nobody waves back.

wearing

Alias	Default text
wearing-already-wearing	{You're} already wearing {the item}.
wearing-already-wearing-there	{You're} already wearing {the item} on {your} {bare part}.
wearing-cant-wear-that	{You} {can't} wear {the item}.
wearing-hands-full	{You} {need} to have {your} hands free to put that on.
wearing-no-target	Wear what?
wearing-not-held	{You} {need} to be holding {the item} first.
wearing-not-wearable	{You} {can't} wear {the item}.
wearing-worn	{You} {put} on {the item}.
wearing-worn-under	{You} would have to take off {the item} first.

Appendix G. Removed forms and their replacements

Forms Chord no longer accepts: 28 of them. The compiler still knows each one by name, so a story that uses one gets an error that says what to write instead. The table gives each form, its replacement and the error's code; the entries under it show a source that raises the error and the compiler's own words. Generated from the `@sharpee/chord` registry of removed forms (Sharpee 5.7.0 · Chord 4.1.0) by `scripts/generate-chord-appendices.cjs`.

Removed form	Write instead	Error
`it` and `its` outside `define trait`	the owner's name	`analysis.it-removed`
a channel's `from event` line	`return <construct> from <event>`	`parse.channel-from-removed`
a channel's `take` line	a `return` line	`parse.channel-take-removed`
`version:`, `blurb:` and `by:` in the story header	`story-version:`, `description:` and `authors:`	`parse.header-unknown-field`
a vehicle's `passage:` line	`pace: <n> turns`	`parse.passage-is-pace`
the one-line `passage <command> to <room>`, `… is blocked` and `… is deadly` lines	a `passage to <room>` block	`parse.passage-line-form`
the phrase strategies `ordered` and `once`	`stopping` and `first-time`	`parse.phrase-strategy-retired`
phrase text on the same line as its key	an indented prose block under the key	`parse.phrase-text-form`

Removed form	Write instead	Error
`define action <name> from …`	a `define action <name>` block with an indented body	`parse.removed-action-hatch`
`define behavior … from`	a `define trait` with clauses, composed on the entity	`parse.removed-behavior-hatch`
`with max items`, `with max weight` and `with capacity` on a kind	a `holds <n>` line	`parse.removed-capacity-setting`
`define chain … from`	nothing yet: Chord has no form for this	`parse.removed-chain-hatch`
`with key the …` and `with tool the …`	`with the <thing>`	`parse.removed-config-keyword`
a region's `containing` line	an `in the <region>` line in each room, or `rooms <name> <first> to <last>`	`parse.removed-containing`
`create the player`	a named `playable` person, given the role in a `before the game starts` block	`parse.removed-create-player`
`define verb`	an `extend action <name>` block with a `grammar` line	`parse.removed-define-verb`
a top-level `every N turns` rule	a `define sequence`, or an `on every turn` clause on the owner	`parse.removed-every`
`define flag`	`states:` on the owner, or a condition over the world	`parse.removed-flag`
the `flag` field type in a trait's data	`states:` on the trait	`parse.removed-flag-field`

Removed form	Write instead	Error
a clause head that ends in `it`, as in `on taking it`	a head that names who acts, as in `on the player taking`	`parse.removed-head-it`
`if` blocks	a `must` line, a `when` suffix on a statement, or `select`	`parse.removed-if`
a top-level `once <condition>` rule	a `, once` modifier on a clause of the owner	`parse.removed-once`
`define score`	a `score <name> worth N` line on the owner or in the story header	`parse.removed-score`
the `(name)` and `:name` slot spellings	`the <name>`	`parse.removed-slot-spelling`
`story "Title" by "Author"`	a bare `story` line with indented `title:` and `authors:` fields	`parse.removed-story-header`
`define text … from`	a `define phrase`, with a strategy if the text varies	`parse.removed-text-hatch`
a top-level `when` rule	an `after the player <verb>` clause on the entity or room it is about	`parse.removed-when`
the `select` strategies `ordered` and `once`	`stopping` and `first-time`	`parse.select-strategy-retired`

analysis.it-removed

```
create the sword
  a thing

  on the player taking
    remove it
  end on
```

`it` is no longer a form outside `define trait` — name the owner: `the sword`.

parse.channel-from-removed

```
define channel clock
  mode replace
  from event estate-clock
end channel
```

`from event` was removed — the event rides the `return … from <event>` clause: `return <construct> from <event>`.

parse.channel-take-removed

```
define channel clock
  mode replace
  take hour
end channel
```

`take` was removed — a channel `return`s a construct: `return <field> from <event>`, `return "text (slot)" from <event>`, or `return phrase <key> from <event>`.

parse.header-unknown-field

```
story
  title: The Lamp
  authors:
    A. Writer
  version: 1.0.0
```

`version:` was removed from the story header — use `story-version:` instead.

parse.passage-is-pace

```
create the gondola
  a vehicle
  passage: 5 turns
```

`passage:` is retired — a flight vehicle's trip time is `pace: <n> turns`, the line every moving vehicle uses.

parse.passage-line-form

```
create the Trench
  a room
  passage climb the rope to the Loft
```

A passage is a block: `passage to <room>`, then `grammar` with one command per line under it, and `end passage`. For example:

```
  passage to the Loft
    grammar
      climb the rope
  end passage
```

parse.phrase-strategy-retired

```
define phrase hint, ordered
  One.
or
  Two.
end phrase
```

`ordered` is no longer a strategy adverb — use `stopping`.

parse.phrase-text-form

```
define phrases en-US
  greet: "Hello."
```

Phrase text goes in an indented prose block — the same-line form was removed.

parse.removed-action-hatch

```
define action juggling from "./stunts.ts"
```

`define action juggling from …` was removed — an action is written in Chord. Write it as a `define action juggling` block with an indented body.

parse.removed-behavior-hatch

```
define behavior glowing from "./glow.ts"
```

`define behavior … from` was removed — it had no binding key and could never fire. Author the behavior in Chord: `define trait <name>` with `on the player <verb>` clauses, composed on the entity.

parse.removed-capacity-setting

```
create the jar
  a container with max items 3
```

`with max items` was removed — write the capacity as its own line: `holds 3`.

parse.removed-chain-hatch

```
define chain opened-revealed from "./reveal.ts"
```

`define chain … from` was removed — a story cannot replace a standard event chain with its own code. Chord has no form for this yet; the behavior needs support in the language.

parse.removed-config-keyword

```
create the gate
  a door
  lockable with key the brass key
```

`with key the …` was removed — write the entity directly: `with the brass key`.

parse.removed-containing

```
create the Cellars
  a region
  containing the Vault
```

`containing` was removed — a region declares identical rooms with `rooms <name> <first> to <last>`, and any other room joins it with an `in the <region>` line in its own block.

parse.removed-create-player

```
create the player
  a person
```

`create the player` was removed — name the character, mark it `playable`, and assign the role in a `before the game starts` block.

parse.removed-define-verb

```
define verb hang means put the item on the hook
```

`define verb` was removed — add the verb onto the action directly: `extend action <name>` with a `grammar` pattern line (e.g. `extend action putting` → `hook the item on the hook`).

parse.removed-every

```
every 3 turns
  phrase drip
end every
```

```
Top-level `every N turns` rules were removed — use a story-owned
`define sequence`, or an every-turn clause on the owner.
```

parse.removed-flag

```
define flag lamp-found
```

```
`define flag` was removed (Chord has no global booleans) — model
the fact as `states:` on its owner, or derive it with a condition
over world state.
```

parse.removed-flag-field

```
define trait sticky
  data
    glue: flag starts false
end trait
```

```
The `flag` field type was removed (Chord has no booleans at any
scope) — declare `states[, reversible]: …` on the trait and name
what the thing IS in each state.
```

parse.removed-head-it

```
create the sword
  a thing

  on taking it
    phrase sharp
  end on
```

```
`on taking it` is no longer a form — name who acts: `on the player
taking` (or the actor's name).
```

parse.removed-if

```
create the Hall
  a room

  on the player entering
    if the lamp is lit
      win
    end if
  end on
```

`if` was removed — use a `must` requirement for guards (`it must be hungry: already-fed`), a statement `when` suffix for conditionals (`award X when <condition>`), or `select` for branching.

parse.removed-once

```
once the lamp is lit
  win
end once
```

Top-level `once <condition>` rules were removed — use a `, once` clause modifier on the owner: `on every turn while <condition>, once`.

parse.removed-score

```
define score treasure worth 10
```

`define score` was removed — attach the score to its earning owner: `score <name> worth N` in the owner's create/trait/action block or the story header.

parse.removed-slot-spelling

```
extend action smelling
  grammar
    sniff (something)
```

```
The `(something)` slot spelling was removed — write `the
something`.
```

parse.removed-story-header

```
story "The Lamp" by "A. Writer"
```

```
`story "Title" by "Author"` was removed — use the fielded form: a
bare `story` line, then indented `title: <title>` and an
`authors:` list (one name per indented line).
```

parse.removed-text-hatch

```
define text garbled from "./extras.ts"
```

```
`define text … from` was removed — a story's text is written in
Chord. Write it as a `define phrase`, with a strategy (`cycling`,
`first-time`, `randomly`, `sticky`, `stopping`) if it varies.
```

parse.removed-when

```
when the player takes the lamp
  win
end when
```

```
Top-level `when` rules were removed — attach the rule to its
owner: `after the player <verb>` on the entity or room it is
about.
```

parse.select-strategy-retired

```
create the lever
  a thing

  on the player pulling
    select ordered
      phrase first-pull
    or
      phrase second-pull
    end select
  end on
```

```
`ordered` is no longer a strategy adverb — use `stopping`.
```