Unfold Studio Documentation

Release 0.4.1

Chris Proctor

Nov 05, 2018

CONTENTS:

Unfold Studio is an online community for interactive storytelling powered by a programming language called Ink. Interactive storytelling brings together the power of programming with the ability of stories to represent and explore our lived realities. Free and open-source, Unfold Studio was developed as part of my PhD research on youth computational literacy practices.

Unfold Studio is used in schools, clubs, and by many individual writers. Interactive storytelling can be a way to integrate Computer Science into English, Social Studies, or other subjects. It can also be an excellent way to introduce Computer Science as a subject relevant to questions of identity, culture, and social justice. (We are currently doing research with a school which uses Unfold Studio for several months as part of its core CS curriculum.)

This documentation is meant for several audiences. If you need help using Unfold Studio or writing interactive stories, see the *User Guide*. (If you're impatient, try the *Quickstart*.) If you are interested in using Unfold Studio with students, see *Teaching Guide*. And if you're interested in Unfold Studio's back story or research on transliteracies, CS education, etc. please see *Research*. We welcome questions, feedback, and random ideas. Please see *Contact* to get in touch.

The documentation is also available in PDF form in case you prefer to read it that way or want to print out any pages (such as the worksheets in the *Teaching Guide* section) for classroom use.

-Chris Proctor

PhD candidate, Stanford Graduate School of Education

Unfold Studio creator and lead researcher

CHAPTER

ONE

WHAT'S NEW

Unfold Studio 0.4.1 includes some exciting new features. We are now working with teachers in several countries and many states across the United States. We would love to hear from you and discuss how Unfold Studio might fit into your teaching. (See *Contact*)

- Upgraded to Ink language version 0.8.2. Along with a bunch of new math functions, this allows the use of lists (See *Part 5: Advanced State Tracking*).
- Stories can now be combined together into very large stories. See *Stories can be combined*.
- Users now have a feed showing them activity on their stories and from users they follow. Your feed appears on your profile page.
- Email-based password reset has been made more reliable.
- Fixed a lot of little bugs.

Version 0.5 will be the next major release. We are targeting the following features:

- Private installations will give teachers and schools access to separate worlds, where content is only visible to logged-in members. Private installations will also come with some teacher-focused features.
- Single sign on integration
- internationalization (Unfold Studio's interface will show up in users' preferred language). Writing stories in multiple languages (and emoji!) is already supported and heartily encouraged.

CHAPTER

TWO

USER GUIDE

2.1 Quickstart

First time here? This will guide you through writing your first story.

Open Unfold Studio in a new tab. Click on **New Story** and then give your story a name. Now copy the code below into your new story and save it. Now you can play it on the right.

```
This is my very first time writing interactive fiction. I am feeling
* excited.
    -> high_energy
* apprehensive.
    -> high_energy
* bored.
    -> low_energy

=== high_energy ===
I hope I can channel this into some quality writing.
    -> DONE

=== low_energy ===
I hope I find something that sparks my curiosity.
    -> DONE
```

This is interactive storytelling. You can write stories that are playable, not just readable. You can give the player as much control as you want.

You probably guessed how the code works. `*` marks choices offered to the player. Each choice has some text and then a divert (`->`), which sends the story off to another *knot*. Knots are defined with headers that look like `=== high_energy ===`. Use `DONE` to show where a story ends. Try adding some choices to one of the knots. Maybe these choices should divert to a new knot…

Now, if you return to the homepage (click **Unfold Studio** in the menu), your story might be featured because it's fresh content. (Stories also get featured when they have a lot of readers, or are loved by a lot of people.) Since you don't have an account yet, it's a public story. Anybody can play it and anybody can edit it.

2.1.1 What's next…

- **Play some stories**. The homepage has featured stories; you can see the rest by clicking **Browse** in the menu.
- **Practice writing more complex stories** with *Story Prompts*.
- **Learn more about the language** at *The Ink language*.
- **Sign up**. Once you have an account, you can create private stories and then share them once you're ready. You can also <3 stories you love, follow other users, collect stories into *books*, and more.

We hope you enjoy Unfold Studio!

2.2 Using Unfold Studio

2.2.1 User accounts

You can participate in Unfold Studio without signing up. Before you have an account, all the stories you write are public. This means anybody can play and edit them. You can also play public stories and stories other authors have shared.

Public stories are good for quick experiments, but once you're ready to write stories that matter to you, you should create an account. When you are logged in to your account, you are the author of your stories and nobody else can edit them.

Unfold Studio is free and open-source; signing up is safe and easy. You will be asked to give your email and choose a username and password. If you forget your password, you can have a reset code sent to your email address.

2.2.2 Writing

When you are logged in, your stories will be private until you choose to share them. Sharing a story makes it visible to everybody.

Todo: Sharing stories, public stories. Private sites.

2.2.3 Reading

Todo: Feeds, follows, loves.

2.2.4 Community

Todo: Norms. It's YA. Not monitored, no guarantee you won't find something horrible.

Note: All different for private installations.

2.3 The Ink language

Note: This page was adapted from Inkle's original documentation for Ink.

2.3.1 Introduction

ink is a scripting language built around the idea of marking up pure-text with flow in order to produce interactive scripts.

At its most basic, it can be used to write a Choose Your Own-style story, or a branching dialogue tree. But its real strength is in writing dialogues with lots of options and lots of recombination of the flow.

ink offers several features to enable non-technical writers to branch often, and play out the consequences of those branches, in both minor and major ways, without fuss.

The script aims to be clean and logically ordered, so branching dialogue can be tested "by eye". The flow is described in a declarative fashion where possible.

It's also designed with redrafting in mind; so editing a flow should be fast.

2.3.2 Part One: The Basics

1) Content

The simplest ink script

The most basic ink script is just text in a .ink file.

```
Hello, world!
```

On running, this will output the content, and then stop.

Text on separate lines produces new paragraphs. The script:

```
Hello, world!
Hello?
Hello, are you there?
```

produces output that looks the same.

Comments

By default, all text in your file will appear in the output content, unless specially marked up.

The simplest mark-up is a comment. **ink** supports two kinds of comment. There's the kind used for someone reading the code, which the compiler ignores:

```
"What do you make of this?" she asked.

// Something unprintable...

"I couldn't possibly comment," I replied.

/*
    ... or an unlimited block of text
*/
```

Tags

Tags don't appear in story flow; instead they can be used to change how the story is presented. Currently, Unfold Studio does not support any tags, but in the future tags will allow you to style content to appear as text messages, Facebook posts, etc.

ink provides a simple system for tagging lines of content, with hashtags.

```
=== content
    A line of normal game-text. # colour it blue
```

2) Choices

Input is offered to the player via text choices. A text choice is indicated by an * character.

If no other flow instructions are given, once made, the choice will flow into the next line of text.

```
Hello world!
*   Hello back!
    Nice to hear from you!
```

This produces the following game:

```
Hello world
1: Hello back!

> 1
Hello back!
Nice to hear from you.
```

By default, the text of a choice appears again, in the output.

Suppressing choice text

Some games separate the text of a choice from its outcome. In **ink**, if the choice text is given in square brackets, the text of the choice will not be printed into response.

```
Hello world!
*   [Hello back!]
    Nice to hear from you!
```

produces

```
Hello world
1: Hello back!

> 1
Nice to hear from you.
```

Advanced: mixing choice and output text

The square brackets in fact divide up the option content. What's before is printed in both choice and output; what's inside only in choice; and what's after, only in output. Effectively, they provide alternative ways for a line to end.

```
Hello world!
*   Hello [back!] right back to you!
    Nice to hear from you!
```

produces:

```
Hello world
1: Hello back!
> 1
Hello right back to you!
Nice to hear from you.
```

This is most useful when writing dialogue choices:

```
"What's that?" my master asked.
*   "I am somewhat tired[."]," I repeated.
    "Really," he responded. "How deleterious."
```

produces:

```
"What's that?" my master asked.
1. "I am somewhat tired."
> 1
"I am somewhat tired," I repeated.
"Really," he responded. "How deleterious."
```

Multiple Choices

To make choices really choices, we need to provide alternatives. We can do this simply by listing them:

```
"What's that?" my master asked.
*   "I am somewhat tired[."]," I repeated.
    "Really," he responded. "How deleterious."
*   "Nothing, Monsieur!"[] I replied.
    "Very good, then."
*  "I said, this journey is appalling[."] and I want no more of it."
    "Ah," he replied, not unkindly. "I see you are feeling frustrated.␣
↪Tomorrow, things will improve."
```

This produces the following game:

```
"What's that?" my master asked.

1: "I am somewhat tired."
```

(continues on next page)

(continued from previous page)

```
2: "Nothing, Monsieur!"
3: "I said, this journey is appalling."

> 3
"I said, this journey is appalling and I want no more of it."
"Ah," he replied, not unkindly. "I see you are feeling frustrated.␣
↪Tomorrow, things will improve."
```

The above syntax is enough to write a single set of choices. In a real game, we'll want to move the flow from one point to another based on what the player chooses. To do that, we need to introduce a bit more structure.

3) Knots

Pieces of content are called knots

To allow the game to branch we need to mark up sections of content with names (as an old-fashioned gamebook does with its 'Paragraph 18', and the like.)

These sections are called "knots" and they're the fundamental structural unit of ink content.

Writing a knot

The start of a knot is indicated by two or more equals signs, as follows.

```
=== top_knot ===
```

(The equals signs on the end are optional; and the name needs to be a single word with no spaces.)

The start of a knot is a header; the content that follows will be inside that knot.

```
=== back_in_london ===

We arrived into London at 9.45pm exactly.
```

Advanced: a knottier "hello world"

When you start an ink file, content outside of knots will be run automatically. But knots won't. So if you start using knots to hold your content, you'll need to tell the game where to go. We do this with a divert arrow `->`, which is covered properly in the next section.

The simplest knotty script is:

```
-> top_knot

=== top_knot ===
Hello world!
```

However, **ink** doesn't like loose ends, and produces a warning on compilation and/or run-time when it thinks this has happened. The script above produces this on compilation:

```
WARNING: Apparent loose end exists where the flow runs out. Do you␣
↪need a '-> END' statement, choice or divert? on line 3 of tests/test.
↪ink
```

and this on running:

```
Runtime error in tests/test.ink line 3: ran out of content. Do you␣
↪need a '-> DONE' or '-> END'?
```

The following plays and compiles without error:

```
=== top_knot ===
Hello world!
-> END
```

`-> END` is a marker for both the writer and the compiler; it means "the story flow should now stop".

4) Diverts

Knots divert to knots

You can tell the story to move from one knot to another using `->`, a "divert arrow". Diverts happen immediately without any user input.

```
=== back_in_london ===

We arrived into London at 9.45pm exactly.
-> hurry_home

=== hurry_home ===
We hurried home to Savile Row as fast as we could.
```

Diverts are invisible

Diverts are intended to be seamless and can even happen mid-sentence:

```
=== hurry_home ===
We hurried home to Savile Row -> as_fast_as_we_could

=== as_fast_as_we_could ===
as fast as we could.
```

produces the same line as above:

```
We hurried home to Savile Row as fast as we could.
```

Glue

The default behaviour inserts line-breaks before every new line of content. In some cases, however, content must insist on not having a line-break, and it can do so using <>, or "glue".

```
=== hurry_home ===
We hurried home <>
-> to_savile_row

=== to_savile_row ===
to Savile Row
-> as_fast_as_we_could

=== as_fast_as_we_could ===
<> as fast as we could.
```

also produces:

```
We hurried home to Savile Row as fast as we could.
```

You can't use too much glue: multiple glues next to each other have no additional effect. (And there's no way to "negate" a glue; once a line is sticky, it'll stick.)

5) Branching The Flow

Basic branching

Combining knots, options and diverts gives us the basic structure of a choose-your-own game.

```
=== paragraph_1 ===
You stand by the wall of Analand, sword in hand.
* [Open the gate] -> paragraph_2
* [Smash down the gate] -> paragraph_3
```

(continues on next page)

(continued from previous page)

```
* [Turn back and go home] -> paragraph_4

=== paragraph_2 ===
You open the gate, and step out onto the path.

...
```

Branching and joining

Using diverts, the writer can branch the flow, and join it back up again, without showing the player that the flow has rejoined.

```
=== back_in_london ===

We arrived into London at 9.45pm exactly.

*   "There is not a moment to lose!"[] I declared.
    -> hurry_outside

*   "Monsieur, let us savour this moment!"[] I declared.
    My master clouted me firmly around the head and dragged me out of␣
↪the door.
    -> dragged_outside

*   [We hurried home] -> hurry_outside


=== hurry_outside ===
We hurried home to Savile Row -> as_fast_as_we_could


=== dragged_outside ===
He insisted that we hurried home to Savile Row
-> as_fast_as_we_could


=== as_fast_as_we_could ===
<> as fast as we could.
```

The story flow

Knots and diverts combine to create the basic story flow of the game. This flow is “flat” - there’s no call-stack, and diverts aren’t “returned” from.

In most ink scripts, the story flow starts at the top, bounces around in a spaghetti-like mess, and eventually, hopefully, reaches a `-> END`.

The very loose structure means writers can get on and write, branching and rejoining without worrying about the structure that they're creating as they go. There's no boiler-plate to creating new branches or diversions, and no need to track any state.

Advanced: Loops

You absolutely can use diverts to create looped content, and **ink** has several features to exploit this, including ways to make the content vary itself, and ways to control how often options can be chosen.

See the sections on Varying Text and Conditional Options for more information.

Oh, and the following is legal and not a great idea:

```
=== round ===
and
-> round
```

6) Includes and Stitches

Knots can be subdivided

As stories get longer, they become more confusing to keep organised without some additional structure.

Knots can include sub-sections called "stitches". These are marked using a single equals sign.

```
=== the_orient_express ===
= in_first_class
    ...
= in_third_class
    ...
= in_the_guards_van
    ...
= missed_the_train
    ...
```

One could use a knot for a scene, for instance, and stitches for the events within the scene.

Stitches have unique names

A stitch can be diverted to using its "address".

```
*   [Travel in third class]
    -> the_orient_express.in_third_class

*   [Travel in the guard's van]
    -> the_orient_express.in_the_guards_van
```

The first stitch is the default

Diverting to a knot which contains stitches will divert to the first stitch in the knot. So:

```
*   [Travel in first class]
    "First class, Monsieur. Where else?"
    -> the_orient_express
```

is the same as:

```
*   [Travel in first class]
    "First class, Monsieur. Where else?"
    -> the_orient_express.in_first_class
```

(. . . unless we move the order of the stitches around inside the knot!)

You can also include content at the top of a knot outside of any stitch. However, you need to remember to divert out of it - the engine *won't* automatically enter the first stitch once it's worked its way through the header content.

```
=== the_orient_express ===

We boarded the train, but where?
*   [First class] -> in_first_class
*   [Second class] -> in_second_class

= in_first_class
    ...
= in_second_class
    ...
```

Local diverts

From inside a knot, you don't need to use the full address for a stitch.

```
-> the_orient_express

```

(continues on next page)

(continued from previous page)

```
=== the_orient_express ===
= in_first_class
    I settled my master.
    *   [Move to third class]
        -> in_third_class

= in_third_class
    I put myself in third.
```

This means stitches and knots can't share names, but several knots can contain the same stitch name. (So both the Orient Express and the SS Mongolia can have first class.)

The compiler will warn you if ambiguous names are used.

Stories can be combined

You can *INCLUDE* other stories on Unfold Studio by referring to their story IDs (visible in the story's URL). Stories must be public or shared to be included in other stories. This can be used to create massive interconnected worlds. Here's a small example which uses a common pattern: the story includes other stories which each describe a particular location, and then re-defines the transition knots (*forest_trail*, *hut_trail*, and *hut_door*) so that the locations are linked up with each other:

```
INCLUDE 1001
INCLUDE 1002

-> forest

=== forest_trail ===
-> hut

=== hut_trail ===
-> forest

=== hut_door ===
You open the door to the hut. To be continued... -> END
```

Story 1001:

```
-> forest

=== forest ===
You are standing in a cool dark forest. There is a faint trail.
+ [Follow the trail] -> forest_trail
```

(continues on next page)

(continued from previous page)

```

=== forest_trail ===
This leads nowhere. -> END
```

Story 1002:

```
// Story 1002
-> hut

=== hut ===
Near the edge of the forest, amongst the smaller trees, you find
a hut. There is smoke coming out of the chimney.
+ [Follow the trail] -> hut_trail
+ [Enter the hut] -> hut_door

=== hut_trail ===
This leads nowhere. -> END

=== hut_trail ===
This leads nowhere. -> END
```

Include statements should always go at the top of a file, and not inside knots. There are no rules about what file a knot must be in to be diverted to. (In other words, separating files has no effect on the game's namespacing).

Here's what happens when you include a story:

- Unfold Studio tries to fetch the story. It must exist, and be either public or shared, and not have any errors. If that story includes others, these are dealt with first.
- Each knot in the included story is added to the story, unless there is already a knot with the same name.
- Each variable declared by the included story is added to beginning of the story, unless there is already a variable with the same name.
- Any action defined outside of a knot in the included story is ignored. This includes changing the values of variables, diverting to knots, showing welcome messages, etc.

This procedure makes it possible to patch certain parts of a story by including it and then redefining just a few knots. It also makes it possible to have stories which can run separately or be included together in larger stories. Here's a live example.

Note: Unfold Studio handles includes differently from regular Ink. See *Include* for details.

Note: Including other authors' writing in your stories creates fascinating collaborative possibilities, but it also means your story isn't fully under your control. If you include a story from someone you don't know, be aware that they could later change the included story. If you want to prevent this, you could fork the other story and include your new copy. Changes to included stories are updated every time you save your story.

5) Varying Choices

Choices can only be used once

By default, every choice in the game can only be chosen once. If you don't have loops in your story, you'll never notice this behaviour. But if you do use loops, you'll quickly notice your options disappearing…

```
=== find_help ===

    You search desperately for a friendly face in the crowd.
    *   The woman in the hat[?] pushes you roughly aside. -> find_help
    *   The man with the briefcase[?] looks disgusted as you stumble␣
 ↪past him. -> find_help
```

produces:

```
You search desperately for a friendly face in the crowd.

1: The woman in the hat?
2: The man with the briefcase?

> 1
The woman in the hat pushes you roughly aside.
You search desperately for a friendly face in the crowd.

1: The man with the briefcase?

>
```

… and on the next loop you'll have no options left.

Fallback choices

The above example stops where it does, because the next choice ends up in an "out of content" run-time error.

```
> 1
The man with the briefcase looks disgusted as you stumble past him.
You search desperately for a friendly face in the crowd.

Runtime error in tests/test.ink line 6: ran out of content. Do you␣
↪need a '-> DONE' or '-> END'?
```

We can resolve this with a ‘fallback choice’. Fallback choices are never displayed to the player, but are ‘chosen’ by the game if no other options exist.

A fallback choice is simply a “choice without choice text”:

```
*   -> out_of_options
```

And, in a slight abuse of syntax, we can make a default choice with content in it, using an “choice then arrow”:

```
*   ->
    Mulder never could explain how he got out of that burning box car.␣
↪-> season_2
```

Example of a fallback choice

Adding this into the previous example gives us:

```
=== find_help ===

    You search desperately for a friendly face in the crowd.
    *   The woman in the hat[?] pushes you roughly aside. -> find_help
    *   The man with the briefcase[?] looks disgusted as you stumble␣
↪past him. -> find_help
    *   ->
        But it is too late: you collapse onto the station platform.␣
↪This is the end.
        -> END
```

and produces:

```
You search desperately for a friendly face in the crowd.

1: The woman in the hat?
2: The man with the briefcase?

> 1
The woman in the hat pushes you roughly aside.
```

(continues on next page)

(continued from previous page)

```
You search desperately for a friendly face in the crowd.

1: The man with the briefcase?

> 1
The man with the briefcase looks disgusted as you stumble past him.
You search desperately for a friendly face in the crowd.
But it is too late: you collapse onto the station platform. This is␣
↪the end.
```

Sticky choices

The 'once-only' behaviour is not always what we want, of course, so we have a second kind of choice: the "sticky" choice. A sticky choice is simply one that doesn't get used up, and is marked by a + bullet.

```
=== homers_couch ===
    +   [Eat another donut]
        You eat another donut. -> homers_couch
    *   [Get off the couch]
        You struggle up off the couch to go and compose epic poetry.
        -> END
```

Default choices can be sticky too.

```
=== conversation_loop
    *   [Talk about the weather] -> chat_weather
    *   [Talk about the children] -> chat_children
    +   -> sit_in_silence_again
```

Conditional Choices

You can also turn choices on and off by hand. **ink** has quite a lot of logic available, but the simplest tests is "has the player seen a particular piece of content".

Every knot/stitch in the game has a unique address (so it can be diverted to), and we use the same address to test if that piece of content has been seen.

```
*   { not visit_paris }      [Go to Paris] -> visit_paris
+   { visit_paris    }       [Return to Paris] -> visit_paris

*   { visit_paris.met_estelle } [ Telephone Mme Estelle ] -> phone_
↪estelle
```

Note that the test `knot_name` is true if *any* stitch inside that knot has been seen.

Note also that conditionals don't override the once-only behaviour of options, so you'll still need sticky options for repeatable choices.

Advanced: multiple conditions

You can use several logical tests on an option; if you do, *all* the tests must all be passed for the option to appear.

```
*   { not visit_paris }      [Go to Paris] -> visit_paris
+   { visit_paris } { not bored_of_paris }
    [Return to Paris] -> visit_paris
```

Advanced: knot/stitch labels are actually read counts

The test:

```
*   {seen_clue} [Accuse Mr Jefferson]
```

is actually testing an *integer* and not a true/false flag. A knot or stitch used this way is actually an integer variable containing the number of times the content at the address has been seen by the player.

If it's non-zero, it'll return true in a test like the one above, but you can also be more specific as well:

```
* {seen_clue > 3} [Flat-out arrest Mr Jefferson]
```

Advanced: more logic

ink supports a lot more logic and conditionality than covered here - see the section on 'variables and logic'.

6) Variable Text

Text can vary

So far, all the content we've seen has been static, fixed pieces of text. But content can also vary at the moment of being printed.

Sequences, cycles and other alternatives

The simplest variations of text are provided by alternatives, which are selected from depending on some kind of rule. **ink** supports several types. Alternatives are written inside `{...}` curly brackets, with elements separated by `|` symbols (vertical divider lines).

These are only useful if a piece of content is visited more than once!

Types of alternatives

Sequences (the default):

A sequence (or a "stopping block") is a set of alternatives that tracks how many times its been seen, and each time, shows the next element along. When it runs out of new content it continues the show the final element.

```
The radio hissed into life. {"Three!"|"Two!"|"One!"|There was the␣
↪white noise racket of an explosion.|But it was just static.}

{I bought a coffee with my five-pound note.|I bought a second coffee␣
↪for my friend.|I didn't have enough money to buy any more coffee.}
```

Cycles (marked with a `&`):

Cycles are like sequences, but they loop their content.

```
It was {&Monday|Tuesday|Wednesday|Thursday|Friday|Saturday|Sunday}␣
↪today.
```

Once-only (marked with a `!`):

Once-only alternatives are like sequences, but when they run out of new content to display, they display nothing. (You can think of a once-only alternative as a sequence with a blank last entry.)

```
He told me a joke. {!I laughed politely.|I smiled.|I grimaced.|I␣
↪promised myself to not react again.}
```

Shuffles (marked with a `~`):

Shuffles produce randomised output.

```
I tossed the coin. {~Heads|Tails}.
```

Features of Alternatives

Alternatives can contain blank elements.

```
I took a step forward. {!||||Then the lights went out. -> eek}
```

Alternatives can be nested.

```
The Ratbear {&{wastes no time and |}swipes|scratches} {&at you|into␣
↪your {&leg|arm|cheek}}.
```

Alternatives can include divert statements.

```
I {waited.|waited some more.|snoozed.|woke up and waited more.|gave up␣
↪and left. -> leave_post_office}
```

They can also be used inside choice text:

```
+   "Hello, {&Master|Monsieur Fogg|you|brown-eyes}!"[] I declared.
```

(. . . with one caveat; you can't start an option's text with a `{`, as it'll look like a conditional.)

(. . . but the caveat has a caveat, if you escape a whitespace `\` before your `{` ink will recognise it as text.)

Examples

Alternatives can be used inside loops to create the appearance of intelligent, state-tracking game-play without particular effort.

Here's a one-knot version of whack-a-mole. Note we use once-only options, and a fallback, to ensure the mole doesn't move around, and the game will always end.

```
=== whack_a_mole ===
    {I heft the hammer.|{~Missed!|Nothing!|No good. Where is he?|Ah-ha!
↪ Got him! -> END}}
    The {&mole|{&nasty|blasted|foul} {&creature|rodent}} is {in here␣
↪somewhere|hiding somewhere|still at large|laughing at me|still␣
↪unwhacked|doomed}. <>
    {!I'll show him!|But this time he won't escape!}
    *    [{&Hit|Smash|Try} top-left]      -> whack_a_mole
    *   [{&Whallop|Splat|Whack} top-right] -> whack_a_mole
    *   [{&Blast|Hammer} middle] -> whack_a_mole
    *   [{&Clobber|Bosh} bottom-left]     -> whack_a_mole
    *   [{&Nail|Thump} bottom-right]      -> whack_a_mole
    *   [] Then you collapse from hunger. The mole has defeated you!
        -> END
```

produces the following 'game':

```
I heft the hammer.
The mole is in here somewhere. I'll show him!

1: Hit top-left
2: Whallop top-right
3: Blast middle
4: Clobber bottom-left
5: Nail bottom-right

> 1
Missed!
The nasty creature is hiding somewhere. But this time he won't escape!

1: Splat top-right
2: Hammer middle
3: Bosh bottom-left
4: Thump bottom-right

> 4
Nothing!
The mole is still at large.
1: Whack top-right
2: Blast middle
3: Clobber bottom-left

> 2
Where is he?
The blasted rodent is laughing at me.
1: Whallop top-right
2: Bosh bottom-left

> 1
Ah-ha! Got him!
```

And here's a bit of lifestyle advice. Note the sticky choice - the lure of the television will never fade:

```
=== turn_on_television ===
I turned on the television {for the first time|for the second␣
↪time|again|once more}, but there was {nothing good on, so I turned␣
↪it off again|still nothing worth watching|even less to hold my␣
↪interest than before|nothing but rubbish|a program about sharks and␣
↪I don't like sharks|nothing on}.
+   [Try it again]          -> turn_on_television
*   [Go outside instead]    -> go_outside_instead
```

(continues on next page)

(continued from previous page)

```
=== go_outside_instead ===
-> END
```

Sneak Preview: Multiline alternatives

ink has another format for making alternatives of varying content blocks, too. See the section on "multiline blocks" for details.

Conditional Text

Text can also vary depending on logical tests, just as options can.

```
{met_blofeld: "I saw him. Only for a moment." }
```

and

```
"His real name was {met_blofeld.learned_his_name: Franz|a secret}."
```

These can appear as separate lines, or within a section of content. They can even be nested, so:

```
{met_blofeld: "I saw him. Only for a moment. His real name was {met_
↪blofeld.learned_his_name: Franz|kept a secret}." | "I missed him.␣
↪Was he particularly evil?" }
```

can produce either:

```
"I saw him. Only for a moment. His real name was Franz."
```

or:

```
"I saw him. Only for a moment. His real name was kept a secret."
```

or:

```
"I missed him. Was he particularly evil?"
```

7) Game Queries

ink provides a few useful 'game level' queries about game state, for use in conditional logic. They're not quite parts of the language, but they're always available, and they can't be edited by the author. In a sense, they're the "standard library functions" of the language.

The convention is to name these in capital letters.

CHOICE_COUNT

CHOICE_COUNT returns the number of options created so far in the current chunk. So for instance.

```
*   {false} Option A
*   {true} Option B
*  {CHOICE_COUNT() == 1} Option C
```

produces two options, B and C. This can be useful for controlling how many options a player gets on a turn.

TURNS_SINCE

TURNS_SINCE returns the number of moves (formally, player inputs) since a particular knot/stitch was last visited.

A value of 0 means "was seen as part of the current chunk". A value of -1 means "has never been seen". Any other positive value means it has been seen that many turns ago.

```
*   {TURNS_SINCE(-> sleeping.intro) > 10} You are feeling tired... ->␣
↪sleeping
*   {TURNS_SINCE(-> laugh) == 0}  You try to stop laughing.
```

Note that the parameter passed to TURNS_SINCE is a "divert target", not simply the knot address itself (because the knot address is a number - the read count - not a location in the story...)

TODO: (requirement of passing -c to the compiler)

Advanced: more queries

You can make your own external functions, though the syntax is a bit different: see the section on functions below.

2.3.3 Part 2: Weave

So far, we've been building branched stories in the simplest way, with "options" that link to "pages".

But this requires us to uniquely name every destination in the story, which can slow down writing and discourage minor branching.

ink has a much more powerful syntax available, designed for simplifying story flows which have an always-forwards direction (as most stories do, and most computer programs don't).

This format is called "weave", and its built out of the basic content/option syntax with two new features: the gather mark, -, and the nesting of choices and gathers.

1) Gathers

Gather points gather the flow back together

Let's go back to the first multi-choice example at the top of this document.

```
"What's that?" my master asked.
    *   "I am somewhat tired[."]," I repeated.
        "Really," he responded. "How deleterious."
    *   "Nothing, Monsieur!"[] I replied.
    *  "I said, this journey is appalling[."] and I want no more of it.
↪"
        "Ah," he replied, not unkindly. "I see you are feeling␣
↪frustrated. Tomorrow, things will improve."
```

In a real game, all three of these options might well lead to the same conclusion - Monsieur Fogg leaves the room. We can do this using a gather, without the need to create any new knots, or add any diverts.

```
"What's that?" my master asked.
    *   "I am somewhat tired[."]," I repeated.
        "Really," he responded. "How deleterious."
    *   "Nothing, Monsieur!"[] I replied.
        "Very good, then."
    *  "I said, this journey is appalling[."] and I want no more of it.
↪"
    "Ah," he replied, not unkindly. "I see you are feeling frustrated.␣
↪Tomorrow, things will improve."

-   With that Monsieur Fogg left the room.
```

This produces the following playthrough:

```
"What's that?" my master asked.

1: "I am somewhat tired."
2: "Nothing, Monsieur!"
3: "I said, this journey is appalling."

> 1
"I am somewhat tired," I repeated.
"Really," he responded. "How deleterious."
With that Monsieur Fogg left the room.
```

Options and gathers form chains of content

We can string these gather-and-branch sections together to make branchy sequences that always run forwards.

```
=== escape ===
I ran through the forest, the dogs snapping at my heels.

    *   I checked the jewels[] were still in my pocket, and the feel of them brought a spring to my step. <>

    *  I did not pause for breath[] but kept on running. <>

    *   I cheered with joy. <>

-   The road could not be much further! Mackie would have the engine running, and then I'd be safe.

    *   I reached the road and looked about[]. And would you believe it?
    *   I should interrupt to say Mackie is normally very reliable[]. He's never once let me down. Or rather, never once, previously to that night.

-   The road was empty. Mackie was nowhere to be seen.
```

This is the most basic kind of weave. The rest of this section details additional features that allow weaves to nest, contain side-tracks and diversions, divert within themselves, and above all, reference earlier choices to influence later ones.

The weave philsophy

Weaves are more than just a convenient encapsulation of branching flow; they're also a way to author more robust content. The `escape` example above has already four possible routes through, and a more complex sequence might have lots and lots more. Using normal diverts, one has to check the links by chasing the diverts from point to point and it's easy for errors to creep in.

With a weave, the flow is guaranteed to start at the top and "fall" to the bottom. Flow errors are impossible in a basic weave structure, and the output text can be easily skim read. That means there's no need to actually test all the branches in game to be sure they work as intended.

Weaves also allow for easy redrafting of choice-points; in particular, it's easy to break a sentence up and insert additional choices for variety or pacing reasons, without having to re-engineer any flow.

2) Nested Flow

The weaves shown above are quite simple, “flat” structures. Whatever the player does, they take the same number of turns to get from top to bottom. However, sometimes certain choices warrant a bit more depth or complexity.

For that, we allow weaves to nest.

This section comes with a warning. Nested weaves are very powerful and very compact, but they can take a bit of getting used to!

Options can be nested

Consider the following scene:

```
-   "Well, Poirot? Murder or suicide?"
*   "Murder!"
*   "Suicide!"
-   Ms. Christie lowered her manuscript a moment. The rest of the␣
↪writing group sat, open-mouthed.
```

The first choice presented is “Murder!” or “Suicide!”. If Poirot declares a suicide, there’s no more to do, but in the case of murder, there’s a follow-up question needed - who does he suspect?

We can add new options via a set of nested sub-choices. We tell the script that these new choices are “part of” another choice by using two asterisks, instead of just one.

```
-   "Well, Poirot? Murder or suicide?"
    *   "Murder!"
        "And who did it?"
        * *     "Detective-Inspector Japp!"
        * *     "Captain Hastings!"
        * *     "Myself!"
    *   "Suicide!"
    -   Mrs. Christie lowered her manuscript a moment. The rest of the␣
↪writing group sat, open-mouthed.
```

(Note that it’s good style to also indent the lines to show the nesting, but the compiler doesn’t mind.)

And should we want to add new sub-options to the other route, we do that in similar fashion.

```
-   "Well, Poirot? Murder or suicide?"
    *   "Murder!"
        "And who did it?"
        * *     "Detective-Inspector Japp!"
        * *     "Captain Hastings!"
```

(continues on next page)

(continued from previous page)

```
        * *     "Myself!"
    *   "Suicide!"
        "Really, Poirot? Are you quite sure?"
        * *     "Quite sure."
        * *     "It is perfectly obvious."
    -   Mrs. Christie lowered her manuscript a moment. The rest of the␣
↪writing group sat, open-mouthed.
```

Now, that initial choice of accusation will lead to specific follow-up questions - but either way, the flow will come back together at the gather point, for Mrs. Christie's cameo appearance.

But what if we want a more extended sub-scene?

Gather points can be nested too

Sometimes, it's not a question of expanding the number of options, but having more than one additional beat of story. We can do this by nesting gather points as well as options.

```
-   "Well, Poirot? Murder or suicide?"
        *   "Murder!"
            "And who did it?"
            * *     "Detective-Inspector Japp!"
            * *     "Captain Hastings!"
            * *     "Myself!"
            - -     "You must be joking!"
            * *     "Mon ami, I am deadly serious."
            * *     "If only..."
        *   "Suicide!"
            "Really, Poirot? Are you quite sure?"
            * *     "Quite sure."
            * *     "It is perfectly obvious."
        -   Mrs. Christie lowered her manuscript a moment. The rest of␣
↪the writing group sat, open-mouthed.
```

If the player chooses the "murder" option, they'll have two choices in a row on their sub-branch - a whole flat weave, just for them.

Advanced: What gathers do

Gathers are hopefully intuitive, but their behaviour is a little harder to put into words: in general, after an option has been taken, the story finds the next gather down that isn't on a lower level, and diverts to it.

The basic idea is this: options separate the paths of the story, and gathers bring them back together. (Hence the name, "weave"!)

You can nest as many levels are you like

Above, we used two levels of nesting; the main flow, and the sub-flow. But there's no limit to how many levels deep you can go.

```
-   "Tell us a tale, Captain!"
    *   "Very well, you sea-dogs. Here's a tale..."
        * *     "It was a dark and stormy night..."
                * * *   "...and the crew were restless..."
                        * * * *  "... and they said to their Captain...
↪"
                                * * * * *       "...Tell us a tale␣
↪Captain!"
    *   "No, it's past your bed-time."
-   To a man, the crew began to yawn.
```

After a while, this sub-nesting gets hard to read and manipulate, so it's good style to divert away to a new stitch if a side-choice goes unwieldy.

But, in theory at least, you could write your entire story as a single weave.

Example: a conversation with nested nodes

Here's a longer example:

```
- I looked at Monsieur Fogg
*   ... and I could contain myself no longer.
    'What is the purpose of our journey, Monsieur?'
    'A wager,' he replied.
    * *     'A wager!'[] I returned.
            He nodded.
            * * *   'But surely that is foolishness!'
            * * *  'A most serious matter then!'
            - - -   He nodded again.
            * * *   'But can we win?'
                    'That is what we will endeavour to find out,' he␣
↪answered.
            * * *   'A modest wager, I trust?'
                    'Twenty thousand pounds,' he replied, quite flatly.
            * * *   I asked nothing further of him then[.], and after␣
↪a final, polite cough, he offered nothing more to me. <>
    * *     'Ah[.'],' I replied, uncertain what I thought.
```

(continues on next page)

(continued from previous page)

```
    - -     After that, <>
*   ... but I said nothing[] and <>
- we passed the day in silence.
- -> END
```

with a couple of possible playthroughs. A short one:

```
I looked at Monsieur Fogg

1: ... and I could contain myself no longer.
2: ... but I said nothing

> 2
... but I said nothing and we passed the day in silence.
```

and a longer one:

```
I looked at Monsieur Fogg

1: ... and I could contain myself no longer.
2: ... but I said nothing

> 1
... and I could contain myself no longer.
'What is the purpose of our journey, Monsieur?'
'A wager,' he replied.

1: 'A wager!'
2: 'Ah.'

> 1
'A wager!' I returned.
He nodded.

1: 'But surely that is foolishness!'
2: 'A most serious matter then!'

> 2
'A most serious matter then!'
He nodded again.

1: 'But can we win?'
2: 'A modest wager, I trust?'
3: I asked nothing further of him then.

> 2
```

(continues on next page)

(continued from previous page)

```
'A modest wager, I trust?'
'Twenty thousand pounds,' he replied, quite flatly.
After that, we passed the day in silence.
```

Hopefully, this demonstrates the philosophy laid out above: that weaves offer a compact way to offer a lot of branching, a lot of choices, but with the guarantee of getting from beginning to end!

3) Tracking a Weave

Sometimes, the weave structure is sufficient. But when it’s not, we need a bit more control.

Weaves are largely unaddressed

By default, lines of content in a weave don’t have an address or label, which means they can’t be diverted to, and they can’t be tested for. In the most basic weave structure, choices vary the path the player takes through the weave and what they see, but once the weave is finished those choices and that path are forgotten.

But should we want to remember what the player has seen, we can - we add in labels where they’re needed using the `(label_name)` syntax.

Gathers and options can be labelled

Gather points at any nested level can be labelled using brackets.

```
-   (top)
```

Once labelled, gather points can be diverted to, or tested for in conditionals, just like knots and stitches. This means you can use previous decisions to alter later outcomes inside the weave, while still keeping all the advantages of a clear, reliable forward-flow.

Options can also be labelled, just like gather points, using brackets. Label brackets come before conditions in the line.

These addresses can be used in conditional tests, which can be useful for creating options unlocked by other options.

```
=== meet_guard ===
The guard frowns at you.

*   (greet) [Greet him]
    'Greetings.'
*   (get_out) 'Get out of my way[.'],' you tell the guard.
```

(continues on next page)

(continued from previous page)

```
-   'Hmm,' replies the guard.

*   {greet}     'Having a nice day?' // only if you greeted him

*   'Hmm?'[] you reply.

*   {get_out} [Shove him aside]      // only if you threatened him
    You shove him sharply. He stares in reply, and draws his sword!
    -> fight_guard          // this route diverts out of the weave

-   'Mff,' the guard replies, and then offers you a paper bag. 'Toffee?
↪'
```

Scope

Inside the same block of weave, you can simply use the label name; from outside the block you need a path, either to a different stitch within the same knot:

```
=== knot ===
= stitch_one
    - (gatherpoint) Some content.
= stitch_two
    *   {stitch_one.gatherpoint} Option
```

or pointing into another knot:

```
=== knot_one ===
-   (gather_one)
    * {knot_two.stitch_two.gather_two} Option

=== knot_two ===
= stitch_two
    - (gather_two)
        *   {knot_one.gather_one} Option
```

Advanced: all options can be labelled

In truth, all content in ink is a weave, even if there are no gathers in sight. That means you can label *any* option in the game with a bracket label, and then reference it using the addressing syntax. In particular, this means you can test *which* option a player took to reach a particular outcome.

```
=== fight_guard ===
...
= throw_something
*   (rock) [Throw rock at guard] -> throw
*   (sand) [Throw sand at guard] -> throw

= throw
You hurl {throw_something.rock:a rock|a handful of sand} at the guard.
```

Advanced: Loops in a weave

Labelling allows us to create loops inside weaves. Here's a standard pattern for asking questions of an NPC.

```
- (opts)
    *   'Can I get a uniform from somewhere?'[] you ask the cheerful␣
↪guard.
        'Sure. In the locker.' He grins. 'Don't think it'll fit you,␣
↪though.'
    *   'Tell me about the security system.'
        'It's ancient,' the guard assures you. 'Old as coal.'
    *   'Are there dogs?'
        'Hundreds,' the guard answers, with a toothy grin. 'Hungry␣
↪devils, too.'
    // We require the player to ask at least one question
    *   {loop} [Enough talking]
        -> done
- (loop)
    // loop a few times before the guard gets bored
    { -> opts | -> opts | }
    He scratches his head.
    'Well, can't stand around talking all day,' he declares.
- (done)
    You thank the guard, and move away.
```

Advanced: diverting to options

Options can also be diverted to: but the divert goes to the output of having chosen that choice, *as though the choice had been chosen*. So the content printed will ignore square bracketed text, and if the option is once-only, it will be marked as used up.

```
- (opts)
*   [Pull a face]
```

(continues on next page)

(continued from previous page)

```
    You pull a face, and the soldier comes at you! -> shove

*   (shove) [Shove the guard aside] You shove the guard to one side,␣
↪but he comes back swinging.

*   {shove} [Grapple and fight] -> fight_the_guard

-   -> opts
```

produces:

```
1: Pull a face
2: Shove the guard aside

> 1
You pull a face, and the soldier comes at you! You shove the guard to␣
↪one side, but he comes back swinging.

1: Grapple and fight

>
```

Advanced: Gathers directly after an option

The following is valid, and frequently useful.

```
*   "Are you quite well, Monsieur?"[] I asked.
    - - (quitewell) "Quite well," he replied.
*   "How did you do at the crossword, Monsieur?"[] I asked.
    -> quitewell
*   I said nothing[] and neither did my Master.
-   We feel into companionable silence once more.
```

Note the level 2 gather point directly below the first option: there's nothing to gather here, really, but it gives us a handy place to divert the second option to.

2.3.4 Part 3: Variables and Logic

So far we've made conditional text, and conditional choices, using tests based on what content the player has seen so far.

ink also supports variables, both temporary and global, storing numerical and content data, or even story flow commands. It is fully-featured in terms of logic, and contains a few additional structures to help keep the often complex logic of a branching story better organised.

1) Global Variables

The most powerful kind of variable, and arguably the most useful for a story, is a variable to store some unique property about the state of the game - anything from the amount of money in the protagonist's pocket, to a value representing the protagonist's state of mind.

This kind of variable is called "global" because it can be accessed from anywhere in the story - both set, and read from. (Traditionally, programming tries to avoid this kind of thing, as it allows one part of a program to mess with another, unrelated part. But a story is a story, and stories are all about consequences: what happens in Vegas rarely stays there.)

Defining Global Variables

Global variables can be defined anywhere, via a `VAR` statement. They should be given an initial value, which defines what type of variable they are - integer, floating point (decimal), content, or a story address.

```
VAR knowledge_of_the_cure = false
VAR players_name = "Emilia"
VAR number_of_infected_people = 521
VAR current_epilogue = -> they_all_die_of_the_plague
```

Using Global Variables

We can test global variables to control options, and provide conditional text, in a similar way to what we have previously seen.

```
=== the_train ===
    The train jolted and rattled. { mood > 0:I was feeling positive␣
↪enough, however, and did not mind the odd bump|It was more than I␣
↪could bear}.
    *   { not knows_about_wager } 'But, Monsieur, why are we␣
↪travelling?'[] I asked.
    *   { knows_about_wager} I contemplated our strange adventure[].␣
↪Would it be possible?
```

Advanced: storing diverts as variables

A "divert" statement is actually a type of value in itself, and can be stored, altered, and diverted to.

```
VAR     current_epilogue = -> everybody_dies

```

(continues on next page)

(continued from previous page)

```
=== continue_or_quit ===
Give up now, or keep trying to save your Kingdom?
*   [Keep trying!]    -> more_hopeless_introspection
*   [Give up]         -> current_epilogue
```

Advanced: Global variables are externally visible

Global variables can be accessed, and altered, from the runtime as well from the story, so provide a good way to communicate between the wider game and the story.

The **ink** layer is often be a good place to store gameplay-variables; there's no save/load issues to consider, and the story itself can react to the current values.

Printing variables

The value of a variable can be printed as content using an inline syntax similar to sequences, and conditional text:

```
VAR friendly_name_of_player = "Jackie"
VAR age = 23

My name is Jean Passepartout, but my friend's call me {friendly_name_
↪of_player}. I'm {age} years old.
```

This can be useful in debugging. For more complex printing based on logic and variables, see the section on functions.

Evaluating strings

It might be noticed that above we refered to variables as being able to contain "content", rather than "strings". That was deliberate, because a string defined in ink can contain ink - although it will always evaluate to a string. (Yikes!)

```
VAR a_colour = ""

~ a_colour = "{~red|blue|green|yellow}"

{a_colour}
```

. . . produces one of red, blue, green or yellow.

Note that once a piece of content like this is evaluated, its value is "sticky". (The quantum state collapses.) So the following:

```
The goon hits you, and sparks fly before you eyes, {a_colour} and {a_
↪colour}.
```

. . . won't produce a very interesting effect. (If you really want this to work, use a text function to print the colour!)

This is also why

```
VAR a_colour = "{~red|blue|green|yellow}"
```

is explicitly disallowed; it would be evaluated on the construction of the story, which probably isn't what you want.

2) Logic

Obviously, our global variables are not intended to be constants, so we need a syntax for altering them.

Since by default, any text in an **ink** script is printed out directly to the screen, we use a markup symbol to indicate that a line of content is intended meant to be doing some numerical work, we use the ~ mark.

The following statements all assign values to variables:

```
=== set_some_variables ===
    ~ knows_about_wager = true
    ~ x = (x * x) - (y * y) + c
    ~ y = 2 * x * y
```

and the following will test conditions:

```
{ x == 1.2 }
{ x / 2 > 4 }
{ y - 1 <= x * x }
```

Mathematics

ink supports the four basic mathematical operations (`+`, `-`, `*` and `/`), as well as `%` (or `mod`), which returns the remainder after integer division.

If more complex operations are required, one can write functions (using recursion if necessary), or call out to external, game-code functions (for anything more advanced).

Advanced: numerical types are implicit

Results of operations - in particular, for division - are typed based on the type of the input. So integer division returns integer, but floating point division returns floating point results.

```
~ x = 2 / 3
~ y = 7 / 3
~ z = 1.2 / 0.5
```

assigns `x` to be 0, `y` to be 2 and `z` to be 2.4.

String queries

Oddly for a text-engine, **ink** doesn't have much in the way of string-handling: it's assumed that any string conversion you need to do will be handled by the game code (and perhaps by external functions.) But we support three basic queries - equality, inequality, and substring (which we call ? for reasons that will become clear in a later chapter).

The following all return true:

```
{ "Yes, please." == "Yes, please." }
{ "No, thank you." != "Yes, please." }
{ "Yes, please" ? "ease" }
```

3) Conditional blocks (if/else)

We've seen conditionals used to control options and story content; **ink** also provides an equivalent of the normal if/else-if/else structure.

A simple 'if'

The if syntax takes its cue from the other conditionals used so far, with the `{...}` syntax indicating that something is being tested.

```
{ x > 0:
    ~ y = x - 1
}
```

Else conditions can be provided:

```
{ x > 0:
    ~ y = x - 1
- else:
```

(continues on next page)

(continued from previous page)

```
    ~ y = x + 1
}
```

Extended if/else if/else blocks

The above syntax is actually a specific case of a more general structure, something like a "switch" statement of another language:

```
{
    - x > 0:
        ~ y = x - 1
    - else:
        ~ y = x + 1
}
```

And using this form we can include 'else-if' conditions:

```
{
    - x == 0:
        ~ y = 0
    - x > 0:
        ~ y = x - 1
    - else:
        ~ y = x + 1
}
```

(Note, as with everything else, the white-space is purely for readability and has no syntactic meaning.)

Switch blocks

And there's also an actual switch statement:

```
{ x:
- 0:    zero
- 1:    one
- 2:    two
- else: lots
}
```

Example: context-relevant content

Note these tests don't have to be variable-based and can use read-counts, just as other conditionals can, and the following construction is quite frequent, as a way of saying "do some content which is relevant to the current game state":

```
=== dream ===
    {
        - visited_snakes && not dream_about_snakes:
            ~ fear++
            -> dream_about_snakes

        - visited_poland && not dream_about_polish_beer:
            ~ fear--
            -> dream_about_polish_beer

        - else:
            // breakfast-based dreams have no effect
            -> dream_about_marmalade
    }
```

The syntax has the advantage of being easy to extend, and prioritise.

Conditional blocks are not limited to logic

Conditional blocks can be used to control story content as well as logic:

```
I stared at Monsieur Fogg.
{ know_about_wager:
    <> "But surely you are not serious?" I demanded.
- else:
    <> "But there must be a reason for this trip," I observed.
}
He said nothing in reply, merely considering his newspaper with as␣
↪much thoroughness as entomologist considering his latest pinned␣
↪addition.
```

You can even put options inside conditional blocks:

```
{ door_open:
    *   I strode out of the compartment[] and I fancied I heard my␣
↪master quietly tutting to himself.           -> go_outside
- else:
    *   I asked permission to leave[] and Monsieur Fogg looked␣
↪surprised.   -> open_door
```

(continues on next page)

(continued from previous page)

```
    *   I stood and went to open the door[]. Monsieur Fogg seemed␣
↪untroubled by this small rebellion. -> open_door
}
```

. . . but note that the lack of weave-syntax and nesting in the above example isn't accidental: to avoid confusing the various kinds of nesting at work, you aren't allowed to include gather points inside conditional blocks.

Multiline blocks

There's one other class of multiline block, which expands on the alternatives system from above. The following are all valid and do what you might expect:

```
// Sequence: go through the alternatives, and stick on last
{ stopping:
    -   I entered the casino.
    -  I entered the casino again.
    -  Once more, I went inside.
}

// Shuffle: show one at random
At the table, I drew a card. <>
{ shuffle:
    -   Ace of Hearts.
    -   King of Spades.
    -   2 of Diamonds.
        'You lose!' crowed the croupier.
        -> leave_casino
}

// Cycle: show each in turn, and then cycle
{ cycle:
    - I held my breath.
    - I waited impatiently.
    - I paused.
}

// Once: show each, once, in turn, until all have been shown
{ once:
    - Would my luck hold?
    - Could I win the hand?
}
```

4) Temporary Variables

Temporary variables are for scratch calculations

Sometimes, a global variable is unwieldy. **ink** provides temporary variables for quick calculations of things.

```
=== near_north_pole ===
    ~ temp number_of_warm_things = 0
    { blanket:
        ~ number_of_warm_things++
    }
    { ear_muffs:
        ~ number_of_warm_things++
    }
    { gloves:
        ~ number_of_warm_things++
    }
    { number_of_warm_things > 2:
        Despite the snow, I felt incorrigibly snug.
    - else:
        That night I was colder than I have ever been.
    }
```

The value in a temporary variable is thrown away after the story leaves the stitch in which it was defined.

Knots and stitches can take parameters

A particularly useful form of temporary variable is a parameter. Any knot or stitch can be given a value as a parameter.

```
*   [Accuse Hasting]
        -> accuse("Hastings")
*   [Accuse Mrs Black]
        -> accuse("Claudia")
*   [Accuse myself]
        -> accuse("myself")

=== accuse(who) ===
    "I accuse {who}!" Poirot declared.
    "Really?" Japp replied. "{who == "myself":You did it?|{who}?}"
    "And why not?" Poirot shot back.
```

. . . and you'll need to use parameters if you want to pass a temporary value from one stitch to another!

Example: a recursive knot definition

Temporary variables are safe to use in recursion (unlike globals), so the following will work.

```
-> add_one_to_one_hundred(0, 1)

=== add_one_to_one_hundred(total, x) ===
    ~ total = total + x
    { x == 100:
        -> finished(total)
    - else:
        -> add_one_to_one_hundred(total, x + 1)
    }

=== finished(total) ===
    "The result is {total}!" you announce.
    Gauss stares at you in horror.
    -> END
```

(In fact, this kind of definition is useful enough that **ink** provides a special kind of knot, called, imaginatively enough, a `function`, which comes with certain restrictions and can return a value. See the section below.)

Advanced: sending divert targets as parameters

Knot/stitch addresses are a type of value, indicated by a `->` character, and can be stored and passed around. The following is therefore legal, and often useful:

```
=== sleeping_in_hut ===
    You lie down and close your eyes.
    -> generic_sleep (-> waking_in_the_hut)

===  generic_sleep (-> waking)
    You sleep perchance to dream etc. etc.
    -> waking

=== waking_in_the_hut
    You get back to your feet, ready to continue your journey.
```

…but note the -> in the *generic_sleep* definition: that's the one case in **ink** where a parameter needs to be typed: because it's too easy to otherwise accidentally do the following:

> **=== sleeping_in_hut ===** You lie down and close your eyes. -> generic_sleep (waking_in_the_hut)

… which sends the read count of *waking_in_the_hut* into the sleeping knot, and then attempts to divert to it.

5) Functions

The use of parameters on knots means they are almost functions in the usual sense, but they lack one key concept - that of the call stack, and the use of return values.

ink includes functions: they are knots, with the following limitations and features:

A function:

- cannot contain stitches
- cannot use diverts or offer choices
- can call other functions
- can include printed content
- can return a value of any type
- can recurse safely

(Some of these may seem quite limiting, but for more story-oriented call-stack-style features, see the section of Tunnels.)

Return values are provided via the `~ return` statement.

Defining and calling functions

To define a function, simply declare a knot to be one:

```
=== function say_yes_to_everything ===
    ~ return true

=== function lerp(a, b, k) ===
    ~ return ((b - a) * k) + a
```

Functions are called by name, and with brackets, even if they have no parameters:

```
~ x = lerp(2, 8, 0.3)

*   {say_yes_to_everything()} 'Yes.'
```

As in any other language, a function, once done, returns the flow to wherever it was called from - and despite not being allowed to divert the flow, functions can still call other functions.

```
=== function say_no_to_nothing ===
    ~ return say_yes_to_everything()
```

Functions don't have to return anything

A function does not need to have a return value, and can simply do something that is worth packaging up:

```
=== function harm(x) ===
    { stamina < x:
        ~ stamina = 0
    - else:
        ~ stamina = stamina - x
    }
```

. . . though remember a function cannot divert, so while the above prevents a negative Stamina value, it won't kill a player who hits zero.

Functions can be called inline

Functions can be called on ~ content lines, but can also be called during a piece of content. In this context, the return value, if there is one, is printed (as well as anything else the function wants to print.) If there is no return value, nothing is printed.

Content is, by default, 'glued in', so the following:

```
Monsieur Fogg was looking {describe_health(health)}.

=== function describe_health(x) ===
{
- x == 100:
    ~ return "spritely"
- x > 75:
    ~ return "chipper"
- x > 45:
    ~ return "somewhat flagging"
- else:
    ~ return "despondent"
}
```

produces:

```
Monsieur Fogg was looking despondent.
```

Examples

For instance, you might include:

```
=== function max(a,b) ===
    { a < b:
        ~ return b
    - else:
        ~ return a
    }

=== function exp(x, e) ===
    // returns x to the power e where e is an integer
    { e <= 0:
        ~ return 1
    - else:
        ~ return x * exp(x, e - 1)
    }
```

Then:

```
The maximum of 2^5 and 3^3 is {max(exp(2,5), exp(3,3))}.
```

produces:

```
The maximum of 2^5 and 3^3 is 32.
```

Example: turning numbers into words

The following example is long, but appears in pretty much every inkle game to date. (Recall that a hyphenated line inside multiline curly braces indicates either "a condition to test" or, if the curly brace began with a variable, "a value to compare against".)

```
=== function print_num(x) ===
{
    - x >= 1000:
        {print_num(x / 1000)} thousand { x mod 1000 > 0:{print_num(x␣
↪mod 1000)}}
    - x >= 100:
        {print_num(x / 100)} hundred { x mod 100 > 0:and {print_num(x␣
↪mod 100)}}
    - x == 0:
        zero
    - else:
        { x >= 20:
            { x / 10:
                - 2: twenty
                - 3: thirty
                - 4: forty
```

(continues on next page)

(continued from previous page)

```
                - 5: fifty
                - 6: sixty
                - 7: seventy
                - 8: eighty
                - 9: ninety
            }
            { x mod 10 > 0:<>-<>}
        }
        { x < 10 || x > 20:
            { x mod 10:
                - 1: one
                - 2: two
                - 3: three
                - 4: four
                - 5: five
                - 6: six
                - 7: seven
                - 8: eight
                - 9: nine
            }
        - else:
            { x:
                - 10: ten
                - 11: eleven
                - 12: twelve
                - 13: thirteen
                - 14: fourteen
                - 15: fifteen
                - 16: sixteen
                - 17: seventeen
                - 18: eighteen
                - 19: nineteen
            }
        }
}
```

which enables us to write things like:

```
~ price = 15

I pulled out {print_num(price)} coins from my pocket and slowly
↪counted them.
"Oh, never mind," the trader replied. "I'll take half." And she took
↪{print_num(price / 2)}, and pushed the rest back over to me.
```

Parameters can be passed by reference

Function parameters can also be passed 'by reference', meaning that the function can actually alter the the variable being passed in, instead of creating a temporary variable with that value.

For instance, most **inkle** stories include the following:

```
=== function alter(ref x, k) ===
    ~ x = x + k
```

Lines such as:

```
~ gold = gold + 7
~ health = health - 4
```

then become:

```
~ alter(gold, 7)
~ alter(health, -4)
```

which are slightly easier to read, and (more usefully) can be done inline for maximum compactness.

```
*   I ate a biscuit[] and felt refreshed. {alter(health, 2)}
*   I gave a biscuit to Monsieur Fogg[] and he wolfed it down most␣
↪undecorously. {alter(foggs_health, 1)}
-   <> Then we continued on our way.
```

Wrapping up simple operations in function can also provide a simple place to put debugging information, if required.

6) Constants

Global Constants

Interactive stories often rely on state machines, tracking what stage some higher level process has reached. There are lots of ways to do this, but the most conveninent is to use constants. Like global variables, global constants should be placed at the top of your story.

Sometimes, it's convenient to define constants to be strings, so you can print them out, for gameplay or debugging purposes.

```
CONST HASTINGS = "Hastings"
CONST POIROT = "Poirot"
CONST JAPP = "Japp"

VAR current_chief_suspect = HASTINGS
```

(continues on next page)

(continued from previous page)

```

=== review_evidence ===
    { found_japps_bloodied_glove:
        ~ current_chief_suspect = POIROT
    }
    Current Suspect: {current_chief_suspect}
```

Sometimes giving them values is useful:

```
CONST PI = 3.14
CONST VALUE_OF_TEN_POUND_NOTE = 10
```

And sometimes the numbers are useful in other ways:

```
CONST LOBBY = 1
CONST STAIRCASE = 2
CONST HALLWAY = 3

CONST HELD_BY_AGENT = -1

VAR secret_agent_location = LOBBY
VAR suitcase_location = HALLWAY

=== report_progress ===
{  secret_agent_location = suitcase_location:
    The secret agent grabs the suitcase!
    ~ suitcase_location = HELD_BY_AGENT

-  secret_agent_location < suitcase_location:
    The secret agent moves forward.
    ~ secret_agent_location++
}
```

Constants are simply a way to allow you to give story states easy-to-understand names.

2.3.5 Part 4: Advanced Flow Control

1) Tunnels

The default structure for **ink** stories is a “flat” tree of choices, branching and joining back together, perhaps looping, but with the story always being “at a certain place”.

But this flat structure makes certain things difficult: for example, imagine a game in which the following interaction can happen:

```
=== crossing_the_date_line ===
*   "Monsieur!"[] I declared with sudden horror. "I have just realised.
↪ We have crossed the international date line!"
-   Monsieur Fogg barely lifted an eyebrow. "I have adjusted for it."
*   I mopped the sweat from my brow[]. A relief!
*   I nodded, becalmed[]. Of course he had!
*  I cursed, under my breath[]. Once again, I had been belittled!
```

…but it can happen at several different places in the story. We don't want to have to write copies of the content for each different place, but when the content is finished it needs to know where to return to. We can do this using parameters:

```
=== crossing_the_date_line(-> return_to) ===
...
-   -> return_to

...

=== outside_honolulu ===
We arrived at the large island of Honolulu.
- (postscript)
    -> crossing_the_date_line(-> done)
- (done)
    -> END

...

=== outside_pitcairn_island ===
The boat sailed along the water towards the tiny island.
- (postscript)
    -> crossing_the_date_line(-> done)
- (done)
    -> END
```

Both of these locations now call and execute the same segment of storyflow, but once finished they return to where they need to go next.

But what if the section of story being called is more complex - what if it spreads across several knots? Using the above, we'd have to keep passing the 'return-to' parameter from knot to knot, to ensure we always knew where to return.

So instead, **ink** integrates this into the language with a new kind of divert, that functions rather like a subroutine, and is called a 'tunnel'.

Tunnels run sub-stories

The tunnel syntax looks like a divert, with another divert on the end:

```
-> crossing_the_date_line ->
```

This means "do the crossing_the_date_line story, then continue from here".

Inside the tunnel itself, the syntax is simplified from the parameterised example: all we do is end the tunnel using the `->->` statement which means, essentially, "go on".

```
=== crossing_the_date_line ===
// this is a tunnel!
...
-   ->->
```

Note that tunnel knots aren't declared as such, so the compiler won't check that tunnels really do end in `->->` statements, except at run-time. So you will need to write carefully to ensure that all the flows into a tunnel really do come out again.

Tunnels can also be chained together, or finish on a normal divert:

> … // this runs the tunnel, then diverts to 'done' -> crossing_the_date_line -> done …
>
> … //this runs one tunnel, then another, then diverts to 'done' -> crossing_the_date_line -> check_foggs_health -> done …

Tunnels can be nested, so the following is valid:

```
=== plains ===
= night_time
    The dark grass is soft under your feet.
    +   [Sleep]
        -> sleep_here -> wake_here -> day_time
= day_time
    It is time to move on.

=== wake_here ===
    You wake as the sun rises.
    +   [Eat something]
        -> eat_something ->
    +   [Make a move]
    -   ->->

=== sleep_here ===
    You lie down and try to close your eyes.
    -> monster_attacks ->
    Then it is time to sleep.
```

(continues on next page)

(continued from previous page)

```
    -> dream ->
    ->->
```

. . . and so on.

Advanced: Tunnels use a call-stack

Tunnels are on a call-stack, so can safely recurse.

2) Threads

So far, everything in ink has been entirely linear, despite all the branching and diverting. But it's actually possible for a writer to 'fork' a story into different sub-sections, to cover more possible player actions.

We call this 'threading', though it's not really threading in the sense that computer scientists mean it: it's more like stitching in new content from various places.

Note that this is definitely an advanced feature: the engineering stories becomes somewhat more complex once threads are involved!

Threads join multiple sections together

Threads allow you to compose sections of content from multiple sources in one go. For example:

```
== thread_example ==
I had a headache; threading is hard to get your head around.
<- conversation
<- walking


== conversation ==
It was a tense moment for Monty and me.
 * "What did you have for lunch today?"[] I asked.
    "Spam and eggs," he replied.
 * "Nice weather, we're having,"[] I said.
    "I've seen better," he replied.
 - -> house

== walking ==
We continued to walk down the dusty road.
 * [Continue walking]
```

(continues on next page)

(continued from previous page)

```
    -> house

== house ==
Before long, we arrived at his house.
-> END
```

It allows multiple sections of story to combined together into a single section:

```
I had a headache; threading is hard to get your head around.
It was a tense moment for Monty and me.
We continued to walk down the dusty road.
1: "What did you have for lunch today?"
2: "Nice weather, we're having,"
3: Continue walking
```

On encountering a thread statement such as `<- conversation`, the compiler will fork the story flow. The first fork considered will run the content at `conversation`, collecting up any options it finds. Once it has run out of flow here it'll then run the other fork.

All the content is collected and shown to the player. But when a choice is chosen, the engine will move to that fork of the story and collapse and discard the others.

Note that global variables are *not* forked, including the read counts of knots and stitches.

Uses of threads

In a normal story, threads might never be needed.

But for games with lots of independent moving parts, threads quickly become essential. Imagine a game in which characters move independently around a map: the main story hub for a room might look like the following:

```
CONST HALLWAY = 1
CONST OFFICE = 2

VAR player_location = HALLWAY
VAR generals_location = HALLWAY
VAR doctors_location = OFFICE

== run_player_location
    {
        - player_location == HALLWAY: -> hallway
    }

== hallway ==
```

(continues on next page)

(continued from previous page)

```
    <- characters_present
    *   [Drawers]   -> examine_drawers
    *   [Wardrobe] -> examine_wardrobe
    *  [Go to Office]   -> go_office
    -   -> run_player_location
= examine_drawers
    // etc...

// Here's the thread, which mixes in dialogue for characters you share␣
↪the room with at the moment.

== characters_present(room)
    { generals_location == player_location:
        <- general_conversation
    }
    { doctors_location == room:
        <- doctor_conversation
    }

== general_conversation
    *   [Ask the General about the bloodied knife]
        "It's a bad business, I can tell you."
    -   -> run_player_location

== doctor_conversation
    *   [Ask the Doctor about the bloodied knife]
        "There's nothing strange about blood, is there?"
    -   -> run_player_location
```

Note in particular, that we need an explicit way to return the player who has gone down a side-thread to return to the main flow. In most cases, threads will either need a parameter telling them where to return to, or they'll need to end the current story section.

When does a side-thread end?

Side-threads end when they run out of flow to process: and note, they collect up options to display later (unlike tunnels, which collect options, display them and follow them until they hit an explicit return, possibly several moves later).

Sometimes a thread has no content to offer - perhaps there is no conversation to have with a character after all, or perhaps we have simply not written it yet. In that case, we must mark the end of the thread explicitly.

If we didn't, the end of content might be a story-bug or a hanging story thread, and we want the compiler to tell us about those.

Using `-> DONE`

So cases where we want to mark the end of a thread, we use `-> DONE`: meaning "the flow intentionally ends here".

Note that we don't need a `-> DONE` if the flow ends with options that fail their conditions. The engine treats this as a valid, intentional, end of flow state.

You do not need a ''-> DONE'' in a thread after an option has been chosen. Once an option is chosen, a thread is no longer a thread - it is simply the normal story flow once more.

Using `-> END` in this case will not end the thread, but the whole story flow. (And this is the real reason for having two different ways to end flow.)

Example: adding the same choice to several places

Threads can be used to add the same choice into lots of different places. When using them this way, it's normal to pass a divert as a parameter, to tell the story where to go after the choice is done.

```
=== outside_the_house
The front step. The house smells. Of murder. And lavender.
- (top)
    <- review_case_notes(-> top)
    *   [Go through the front door]
        I stepped inside the house.
        -> the_hallway
    *   [Sniff the air]
        I hate lavender. It makes me think of soap, and soap makes me␣
↪think about my marriage.
        -> top

=== the_hallway
The hallway. Front door open to the street. Little bureau.
- (top)
    <- review_case_notes(-> top)
    *   [Go through the front door]
        I stepped out into the cool sunshine.
        -> outside_the_house
    *   [Open the bureau]
        Keys. More keys. Even more keys. How many locks do these␣
↪people need?
        -> top

=== review_case_notes(-> go_back_to)
+   {not done || TURNS_SINCE(-> done) > 10}
```

(continues on next page)

(continued from previous page)

```
    [Review my case notes]
    // the conditional ensures you don't get the option to check␣
↪repeatedly
    {I|Once again, I} flicked through the notes I'd made so far. Still␣
↪not obvious suspects.
-   (done) -> go_back_to
```

Note this is different than a tunnel, which runs the same block of content but doesn't give a player a choice. So a layout like:

```
<- childhood_memories(-> next)
*   [Look out of the window]
    I daydreamed as we rolled along...
 - (next) Then the whistle blew...
```

might do exactly the same thing as:

```
*   [Remember my childhood]
    -> think_back ->
*   [Look out of the window]
    I daydreamed as we rolled along...
-   (next) Then the whistle blew...
```

but as soon as the option being threaded in includes multiple choices, or conditional logic on choices (or any text content, of course!), the thread version becomes more practical.

Example: organisation of wide choice points

A game which uses ink as a script rather than a literal output might often generate very large numbers of parallel choices, intended to be filtered by the player via some other in-game interaction - such as walking around an environment. Threads can be useful in these cases simply to divide up choices.

```
=== the_kitchen
- (top)
    <- drawers(-> top)
    <- cupboards(-> top)
    <- room_exits
= drawers (-> goback)
    // choices about the drawers...
    ...
= cupboards(-> goback)
    // choices about cupboards
    ...
```

(continues on next page)

(continued from previous page)

```
= room_exits
    // exits; doesn't need a "return point" as if you leave, you go␣
↪elsewhere
    ...
```

2.3.6 Part 5: Advanced State Tracking

Games with lots of interaction can get very complex, very quickly and the writer's job is often as much about maintaining continuity as it is about content.

This becomes particularly important if the game text is intended to model anything - whether it's a game of cards, the player's knowledge of the gameworld so far, or the state of the various light-switches in a house.

ink does not provide a full world-modelling system in the manner of a classic parser IF authoring language - there are no "objects", no concepts of "containment" or being "open" or "locked". However, it does provide a simple yet powerful system for tracking state-changes in a very flexible way, to enable writers to approximate world models where necessary.

Note: This feature is very new to the language. That means we haven't begun to discover all the ways it might be used - but we're pretty sure it's going to be useful! So if you think of a clever usage we'd love to know!

1) Basic Lists

The basic unit of state-tracking is a list of states, defined using the `LIST` keyword. Note that a list is really nothing like a C# list (which is an array).

For instance, we might have:

```
LIST kettleState = cold, boiling, recently_boiled
```

This line defines two things: firstly three new values - `cold`, `boiling` and `recently_boiled` - and secondly, a variable, called `kettleState`, to hold these states.

We can tell the list what value to take:

```
~ kettleState = cold
```

We can change the value:

```
*   [Turn on kettle]
    The kettle begins to bubble and boil.
    ~ kettleState = boiling
```

We can query the value:

```
*   [Touch the kettle]
    { kettleState == cold:
        The kettle is cool to the touch.
    - else:
        The outside of the kettle is very warm!
    }
```

For convenience, we can give a list a value when it's defined using a bracket:

```
LIST kettleState = cold, (boiling), recently_boiled
// at the start of the game, this kettle is switched on. Edgy, huh?
```

. . . and if the notation for that looks a bit redundant, there's a reason for that coming up in a few subsections time.

2) Reusing Lists

The above example is fine for the kettle, but what if we have a pot on the stove as well? We can then define a list of states, but put them into variables - and as many variables as we want.

```
LIST daysOfTheWeek = Monday, Tuesday, Wednesday, Thursday, Friday
VAR today = Monday
VAR tomorrow = Tuesday
```

States can be used repeatedly

This allows us to use the same state machine in multiple places.

```
LIST heatedWaterStates = cold, boiling, recently_boiled
VAR kettleState = cold
VAR potState = cold

*   {kettleState == cold} [Turn on kettle]
    The kettle begins to boil and bubble.
    ~ kettleState = boiling
*   {potState == cold} [Light stove]
    The water in the pot begins to boil and bubble.
    ~ potState = boiling
```

But what if we add a microwave as well? We might want start generalising our functionality a bit:

```
LIST heatedWaterStates = cold, boiling, recently_boiled
VAR kettleState = cold
VAR potState = cold
VAR microwaveState = cold

=== function boilSomething(ref thingToBoil, nameOfThing)
    The {nameOfThing} begins to heat up.
    ~ thingToBoil = boiling

=== do_cooking
*   {kettleState == cold} [Turn on kettle]
    {boilSomething(kettleState, "kettle")}
*   {potState == cold} [Light stove]
    {boilSomething(potState, "pot")}        *   {microwaveState ==␣
↪cold} [Turn on microwave]
    {boilSomething(microwaveState, "microwave")}
```

or even. . .

```
LIST heatedWaterStates = cold, boiling, recently_boiled
VAR kettleState = cold
VAR potState = cold
VAR microwaveState = cold

=== cook_with(nameOfThing, ref thingToBoil)
+   {thingToBoil == cold} [Turn on {nameOfThing}]
    The {nameOfThing} begins to heat up.
    ~ thingToBoil = boiling
    -> do_cooking.done

=== do_cooking
<- cook_with("kettle", kettleState)
<- cook_with("pot", potState)
<- cook_with("microwave", microwaveState)
- (done)
```

Note that the "heatedWaterStates" list is still available as well, and can still be tested, and take a value.

List values can share names

Reusing lists brings with it ambiguity. If we have:

```
LIST colours = red, green, blue, purple
LIST moods = mad, happy, blue

VAR status = blue
```

. . . how can the compiler know which blue you meant?

We resolve these using a `.` syntax similar to that used for knots and stitches.

```
VAR status = colours.blue
```

. . . and the compiler will issue an error until you specify.

Note the "family name" of the state, and the variable containing a state, are totally separate. So

```
{ statesOfGrace == statesOfGrace.fallen:
    // is the current state "fallen"
}
```

. . . is correct.

Advanced: a LIST is actually a variable

One surprising feature is the statement

```
LIST statesOfGrace = ambiguous, saintly, fallen
```

actually does two things simultaneously: it creates three values, `ambiguous`, `saintly` and `fallen`, and gives them the name-parent `statesOfGrace` if needed; and it creates a variable called `statesOfGrace`.

And that variable can be used like a normal variable. So the following is valid, if horribly confusing and a bad idea:

```
LIST statesOfGrace = ambiguous, saintly, fallen

~ statesOfGrace = 3.1415 // set the variable to a number not a list␣
↪value
```

. . . and it wouldn't preclude the following from being fine:

~ temp anotherStateOfGrace = statesOfGrace.saintly

3) List Values

When a list is defined, the values are listed in an order, and that order is considered to be significant. In fact, we can treat these values as if they *were* numbers. (That is to say, they are enums.)

```
LIST volumeLevel = off, quiet, medium, loud, deafening
VAR lecturersVolume = quiet
VAR murmurersVolume = quiet

{ lecturersVolume < deafening:
    ~ lecturersVolume++

    { lecturersVolume > murmurersVolume:
        ~ murmurersVolume++
        The murmuring gets louder.
    }
}
```

The values themselves can be printed using the usual `{...}` syntax, but this will print their name.

```
The lecturer's voice becomes {lecturersVolume}.
```

Converting values to numbers

The numerical value, if needed, can be got explicitly using the LIST_VALUE function. Note the first value in a list has the value 1, and not the value 0.

```
The lecturer has {LIST_VALUE(deafening) - LIST_VALUE(lecturersVolume)}␣
↪notches still available to him.
```

Converting numbers to values

You can go the other way by using the list's name as a function:

```
LIST Numbers = one, two, three
VAR score = one
~ score = Numbers(2) // score will be "two"
```

Advanced: defining your own numerical values

By default, the values in a list start at 1 and go up by one each time, but you can specify your own values if you need to.

```
LIST primeNumbers = two = 2, three = 3, five = 5
```

If you specify a value, but not the next value, ink will assume an increment of 1. So the following is the same:

```
LIST primeNumbers = two = 2, three, five = 5
```

4) Multivalued Lists

The following examples have all included one deliberate untruth, which we'll now remove. Lists - and variables containing list values - do not have to contain only one value.

Lists are boolean sets

A list variable is not a variable containing a number. Rather, a list is like the in/out nameboard in an accommodation block. It contains a list of names, each of which has a room-number associated with it, and a slider to say "in" or "out".

Maybe no one is in:

```
LIST DoctorsInSurgery = Adams, Bernard, Cartwright, Denver, Eamonn
```

Maybe everyone is:

```
LIST DoctorsInSurgery = (Adams), (Bernard), (Cartwright), (Denver),␣
↪(Eamonn)
```

Or maybe some are and some aren't:

```
LIST DoctorsInSurgery = (Adams), Bernard, (Cartwright), Denver, Eamonn
```

Names in brackets are included in the initial state of the list.

Note that if you're defining your own values, you can place the brackets around the whole term or just the name:

```
LIST primeNumbers = (two = 2), (three) = 3, (five = 5)
```

Assiging multiple values

We can assign all the values of the list at once as follows:

```
~ DoctorsInSurgery = (Adams, Bernard)
~ DoctorsInSurgery = (Adams, Bernard, Eamonn)
```

We can assign the empty list to clear a list out:

```
~ DoctorsInSurgery = ()
```

Adding and removing entries

List entries can be added and removed, singly or collectively.

```
~ DoctorsInSurgery = DoctorsInSurgery + Adams   ~ DoctorsInSurgery +=␣
↪Adams  // this is the same as the above
~ DoctorsInSurgery -= Eamonn
~ DoctorsInSurgery += (Eamonn, Denver)
~ DoctorsInSurgery -= (Adams, Eamonn, Denver)
```

Trying to add an entry that's already in the list does nothing. Trying to remove an entry that's not there also does nothing. Neither produces an error, and a list can never contain duplicate entries.

Basic Queries

We have a few basic ways of getting information about what's in a list:

```
LIST DoctorsInSurgery = (Adams), Bernard, (Cartwright), Denver, Eamonn

{LIST_COUNT(DoctorsInSurgery)}  //  "2"
{LIST_MIN(DoctorsInSurgery)}        //  "Adams"
{LIST_MAX(DoctorsInSurgery)}        //  "Cartwright"
```

Testing for emptiness

Like most values in ink, a list can be tested "as it is", and will return true, unless it's empty.

```
{ DoctorsInSurgery: The surgery is open today. | Everyone has gone␣
↪home. }
```

Testing for exact equality

Testing multi-valued lists is slightly more complex than single-valued ones. Equality (==) now means 'set equality' - that is, all entries are identical.

So one might say:

```
{ DoctorsInSurgery == (Adams, Bernard):
    Dr Adams and Dr Bernard are having a loud argument in one corner.
}
```

If Dr Eamonn is in as well, the two won't argue, as the lists being compared won't be equal - DoctorsInSurgery will have an Eamonn that the list (Adams, Bernard) doesn't have.

Not equals works as expected:

```
{ DoctorsInSurgery != (Adams, Bernard):
    At least Adams and Bernard aren't arguing.
}
```

Testing for containment

What if we just want to simply ask if Adams and Bernard are present? For that we use a new operator, `has`, otherwise known as `?`.

```
{ DoctorsInSurgery ? (Adams, Bernard):
    Dr Adams and Dr Bernard are having a hushed argument in one corner.
}
```

And `?` can apply to single values too:

```
{ DoctorsInSurgery has Eamonn:
    Dr Eamonn is polishing his glasses.
}
```

We can also negate it, with `hasnt` or `!?` (not `?`). Note this starts to get a little complicated as

```
DoctorsInSurgery !? (Adams, Bernard)
```

does not mean neither Adams nor Bernard is present, only that they are not *both* present (and arguing).

Example: basic knowledge tracking

The simplest use of a multi-valued list is for tracking "game flags" tidily.

```
LIST Facts = (Fogg_is_fairly_odd),  first_name_phileas, (Fogg_is_
↪English)

```

(continues on next page)

(continued from previous page)

```
{Facts ? Fogg_is_fairly_odd:I smiled politely.|I frowned. Was he a␣
↪lunatic?}
'{Facts ? first_name_phileas:Phileas|Monsieur}, really!' I cried.
```

In particular, it allows us to test for multiple game flags in a single line.

```
{ Facts ? (Fogg_is_English, Fogg_is_fairly_odd):
    <> 'I know Englishmen are strange, but this is *incredible*!'
}
```

Example: a doctor's surgery

We're overdue a fuller example, so here's one.

```
LIST DoctorsInSurgery = (Adams), Bernard, Cartwright, (Denver), Eamonn

-> waiting_room

=== function whos_in_today()
    In the surgery today are {DoctorsInSurgery}.

=== function doctorEnters(who)
    { DoctorsInSurgery !? who:
        ~ DoctorsInSurgery += who
        Dr {who} arrives in a fluster.
    }

=== function doctorLeaves(who)
    { DoctorsInSurgery ? who:
        ~ DoctorsInSurgery -= who
        Dr {who} leaves for lunch.
    }

=== waiting_room
    {whos_in_today()}
    *   [Time passes...]
        {doctorLeaves(Adams)} {doctorEnters(Cartwright)}
↪{doctorEnters(Eamonn)}
        {whos_in_today()}
```

This produces:

```
In the surgery today are Adams, Denver.
```

(continues on next page)

(continued from previous page)

```
> Time passes...

Dr Adams leaves for lunch. Dr Cartwright arrives in a fluster. Dr␣
↪Eamonn arrives in a fluster.

In the surgery today are Cartwright, Denver, Eamonn.
```

Advanced: nicer list printing

The basic list print is not especially attractive for use in-game. The following is better:

```
=== function listWithCommas(list, if_empty)
    {LIST_COUNT(list):
    - 2:
            {LIST_MIN(list)} and {listWithCommas(list - LIST_MIN(list),
↪ if_empty)}
    - 1:
            {list}
    - 0:
            {if_empty}
    - else:
            {LIST_MIN(list)}, {listWithCommas(list - LIST_MIN(list),␣
↪if_empty)}
    }

LIST favouriteDinosaurs = (stegosaurs), brachiosaur, (anklyosaurus),␣
↪(pleiosaur)

My favourite dinosaurs are {listWithCommas(favouriteDinosaurs, "all␣
↪extinct")}.
```

It's probably also useful to have an is/are function to hand:

```
=== function isAre(list)
    {LIST_COUNT(list) == 1:is|are}

My favourite dinosaurs {isAre(favouriteDinosaurs)}
↪{listWithCommas(favouriteDinosaurs, "all extinct")}.
```

And to be pendantic:

```
My favourite dinosaur{LIST_COUNT(favouriteDinosaurs) != 1:s}
↪{isAre(favouriteDinosaurs)} {listWithCommas(favouriteDinosaurs, "all␣
↪extinct")}.
```

Lists don't need to have multiple entries

Lists don't *have* to contain multiple values. If you want to use a list as a state-machine, the examples above will all work - set values using =, ++ and --; test them using ==, <, <=, > and >=. These will all work as expected.

The "full" list

Note that `LIST_COUNT`, `LIST_MIN` and `LIST_MAX` are refering to who's in/out of the list, not the full set of *possible* doctors. We can access that using

```
LIST_ALL(element of list)
```

or

```
LIST_ALL(list containing elements of a list)

{LIST_ALL(DoctorsInSurgery)} // Adams, Bernard, Cartwright, Denver,␣
↪Eamonn
{LIST_COUNT(LIST_ALL(DoctorsInSurgery))} // "5"
{LIST_MIN(LIST_ALL(Eamonn))}                // "Adams"
```

Note that printing a list using `{...}` produces a bare-bones representation of the list; the values as words, delimited by commas.

Advanced: "refreshing" a list's type

If you really need to, you can make an empty list that knows what type of list it is.

```
LIST ValueList = first_value, second_value, third_value
VAR myList = ()

~ myList = ValueList()
```

You'll then be able to do:

```
{ LIST_ALL(myList) }
```

Advanced: a portion of the "full" list

You can also retrieve just a "slice" of the full list, using the `LIST_RANGE` function.

```
LIST_RANGE(list_name, min_value, max_value)
```

Example: Tower of Hanoi

To demonstrate a few of these ideas, here's a functional Tower of Hanoi example, written so no one else has to write it.

```
LIST Discs = one, two, three, four, five, six, seven
VAR post1 = ()
VAR post2 = ()
VAR post3 = ()

~ post1 = LIST_ALL(Discs)

-> gameloop

=== function can_move(from_list, to_list) ===
    {
    -   LIST_COUNT(from_list) == 0:
        // no discs to move
        ~ return false
    -   LIST_COUNT(to_list) > 0 && LIST_MIN(from_list) > LIST_MIN(to_
↪list):
        // the moving disc is bigger than the smallest of the discs on␣
↪the new tower
        ~ return false
    -   else:
         // nothing stands in your way!
        ~ return true

    }

=== function move_ring( ref from, ref to ) ===
    ~ temp whichRingToMove = LIST_MIN(from)
    ~ from -= whichRingToMove
    ~ to += whichRingToMove

== function getListForTower(towerNum)
    { towerNum:
        - 1:    ~ return post1
        - 2:    ~ return post2
        - 3:    ~ return post3
    }
```

(continues on next page)

(continued from previous page)

```
=== function name(postNum)
    the {postToPlace(postNum)} temple

=== function Name(postNum)
    The {postToPlace(postNum)} temple

=== function postToPlace(postNum)
    { postNum:
        - 1: first
        - 2: second
        - 3: third
    }

=== function describe_pillar(listNum) ==
    ~ temp list = getListForTower(listNum)
    {
    - LIST_COUNT(list) == 0:
        {Name(listNum)} is empty.
    - LIST_COUNT(list) == 1:
        The {list} ring lies on {name(listNum)}.
    - else:
        On {name(listNum)}, are the discs numbered {list}.
    }


=== gameloop
    Staring down from the heavens you see your followers finishing␣
↪construction of the last of the great temples, ready to begin the␣
↪work.
- (top)
    +  (describe) {true || TURNS_SINCE(-> describe) >= 2 || !describe}␣
↪[ Regard the temples]
        You regard each of the temples in turn. On each is stacked the␣
↪rings of stone. {describe_pillar(1)} {describe_pillar(2)} {describe_
↪pillar(3)}
    <- move_post(1, 2, post1, post2)
    <- move_post(2, 1, post2, post1)
    <- move_post(1, 3, post1, post3)
    <- move_post(3, 1, post3, post1)
    <- move_post(3, 2, post3, post2)
    <- move_post(2, 3, post2, post3)
    -> DONE

= move_post(from_post_num, to_post_num, ref from_post_list, ref to_
↪post_list)
```

(continues on next page)

(continued from previous page)

```
    +    { can_move(from_post_list, to_post_list) }
         [ Move a ring from {name(from_post_num)} to {name(to_post_num)}
↪ ]
         { move_ring(from_post_list, to_post_list) }
         { stopping:
         -    The priests far below construct a great harness, and after␣
↪many years of work, the great stone ring is lifted up into the air,␣
↪and swung over to the next of the temples.
              The ropes are slashed, and in the blink of an eye it falls␣
↪once more.
         -    Your next decree is met with a great feast and many␣
↪sacrifices. After the funeary smoke has cleared, work to shift the␣
↪great stone ring begins in earnest. A generation grows and falls,␣
↪and the ring falls into its ordained place.
         -    {cycle:
              - Years pass as the ring is slowly moved.
              - The priests below fight a war over what colour robes to␣
↪wear, but while they fall and die, the work is still completed.
              }
         }
    -> top
```

5) Advanced List Operations

The above section covers basic comparisons. There are a few more powerful features as well, but - as anyone familiar with mathematical sets will know - things begin to get a bit fiddly. So this section comes with an 'advanced' warning.

A lot of the features in this section won't be necessary for most games.

Comparing lists

We can compare lists less than exactly using >, <, >= and <=. Be warned! The definitions we use are not exactly standard fare. They are based on comparing the numerical value of the elements in the lists being tested.

"Distinctly bigger than"

`LIST_A > LIST_B` means "the smallest value in A is bigger than the largest values in B": in other words, if put on a number line, the entirety of A is to the right of the entirety of B. < does the same in reverse.

"Definitely never smaller than"

`LIST_A >= LIST_B` means - take a deep breath now - "the smallest value in A is at least the smallest value in B, and the largest value in A is at least the largest value in B". That is, if drawn on a number line, the entirety of A is either above B or overlaps with it, but B does not extend higher than A.

Note that `LIST_A > LIST_B` implies `LIST_A != LIST_B`, and `LIST_A >= LIST_B` allows `LIST_A == LIST_B` but precludes `LIST_A < LIST_B`, as you might hope.

Health warning!

`LIST_A >= LIST_B` is *not* the same as `LIST_A > LIST_B or LIST_A == LIST_B`.

The moral is, don't use these unless you have a clear picture in your mind.

Inverting lists

A list can be "inverted", which is the equivalent of going through the accommodation in/out nameboard and flipping every switch to the opposite of what it was before.

```
LIST GuardsOnDuty = (Smith), (Jones), Carter, Braithwaite

=== function changingOfTheGuard
    ~ GuardsOnDuty = LIST_INVERT(GuardsOnDuty)
```

Note that `LIST_INVERT` on an empty list will return a null value, if the game doesn't have enough context to know what invert. If you need to handle that case, it's safest to do it by hand:

```
=== function changingOfTheGuard
    {!GuardsOnDuty: // "is GuardsOnDuty empty right now?"
        ~ GuardsOnDuty = LIST_ALL(Smith)
    - else:
        ~ GuardsOnDuty = LIST_INVERT(GuardsOnDuty)
    }
```

Footnote

The syntax for inversion was originally `~ list` but we changed it because otherwise the line

```
~ list = ~ list
```

was not only functional, but actually caused list to invert itself, which seemed excessively perverse.

Intersecting lists

The `has` or `?` operator is, somewhat more formally, the superset operator. "A has B" is true when every item in B is also in A, including when they are the same set.

To test for "some overlap" between lists, we use the overlap operator, `^`, to get the *intersection*.

```
LIST CoreValues = strength, courage, compassion, greed, nepotism, self_
↪belief, delusions_of_godhood
VAR desiredValues = (strength, courage, compassion, self_belief )
VAR actualValues =  ( greed, nepotism, self_belief, delusions_of_
↪godhood )

{desiredValues ^ actualValues} // prints "self_belief"
```

The result is a new list, so you can test it:

```
{desiredValues ^ actualValues: The new president has at least one␣
↪desirable quality.}

{LIST_COUNT(desiredValues ^ actualValues) == 1: Correction, the new␣
↪president has only one desirable quality. {desiredValues ^␣
↪actualValues == self_belief: It's the scary one.}}
```

6) Multi-list Lists

So far, all of our examples have included one large simplification, again - that the values in a list variable have to all be from the same list family. But they don't.

This allows us to use lists - which have so far played the role of state-machines and flag-trackers - to also act as general properties, which is useful for world modelling.

This is our inception moment. The results are powerful, but also more like "real code" than anything that's come before.

Lists to track objects

For instance, we might define:

```
LIST Characters = Alfred, Batman, Robin
LIST Props = champagne_glass, newspaper

VAR BallroomContents = (Alfred, Batman, newspaper)
VAR HallwayContents = (Robin, champagne_glass)
```

We could then describe the contents of any room by testing its state:

```
=== function describe_room(roomState)
    { roomState ? Alfred: Alfred is here, standing quietly in a corner.
↪ } { roomState ? Batman: Batman's presence dominates all. } {␣
↪roomState ? Robin: Robin is all but forgotten. }
    <> { roomState ? champagne_glass: A champagne glass lies discarded␣
↪on the floor. } { roomState ? newspaper: On one table, a headline␣
↪blares out WHO IS THE BATMAN? AND *WHO* IS HIS BARELY-REMEMBERED␣
↪ASSISTANT? }
```

So then:

```
{ describe_room(BallroomContents) }
```

produces:

```
Alfred is here, standing quietly in a corner. Batman's presence␣
↪dominates all.

On one table, a headline blares out WHO IS THE BATMAN? AND *WHO* IS␣
↪HIS BARELY-REMEMBERED ASSISTANT?
```

While:

```
{ describe_room(HallwayContents) }
```

gives:

```
Robin is all but forgotten.

A champagne glass lies discarded on the floor.
```

And we could have options based on combinations of things:

```
*   { currentRoomState ? (Batman, Alfred) } [Talk to Alfred and Batman]
    'Say, do you two know each other?'
```

Lists to track multiple states

We can model devices with multiple states. Back to the kettle again…

```
LIST OnOff = on, off
LIST HotCold = cold, warm, hot

VAR kettleState = off, cold

```

(continues on next page)

(continued from previous page)

```
=== function turnOnKettle() ===
{ kettleState ? hot:
    You turn on the kettle, but it immediately flips off again.
- else:
    The water in the kettle begins to heat up.
    ~ kettleState -= off
    ~ kettleState += on
    // note we avoid "=" as it'll remove all existing states
}

=== function can_make_tea() ===
    ~ return kettleState ? (hot, off)
```

These mixed states can make changing state a bit trickier, as the off/on above demonstrates, so the following helper function can be useful.

```
  === function changeStateTo(ref stateVariable, stateToReach)
      // remove all states of this type
      ~ stateVariable -= LIST_ALL(stateToReach)
      // put back the state we want
      ~ stateVariable += stateToReach


which enables code like:
```

```
~ changeState(kettleState, on)
~ changeState(kettleState, warm)
```

How does this affect queries?

The queries given above mostly generalise nicely to multi-valued lists

```
LIST Letters = a,b,c
LIST Numbers = one, two, three

VAR mixedList = (a, three, c)

{LIST_ALL(mixedList)}   // a, one, b, two, c, three
{LIST_COUNT(mixedList)} // 3
{LIST_MIN(mixedList)}   // a
{LIST_MAX(mixedList)}   // three or c, albeit unpredictably

{mixedList ? (a,b) }        // false
```

(continues on next page)

(continued from previous page)

```
{mixedList ^ LIST_ALL(a)}   // a, c

{ mixedList >= (one, a) }   // true
{ mixedList < (three) }     // false

{ LIST_INVERT(mixedList) }            // one, b, two
```

7) Long example: crime scene

Finally, here's a long example, demonstrating a lot of ideas from this section in action. You might want to try playing it before reading through to better understand the various moving parts.

```
-> murder_scene

//
//  System: items can have various states
//  Some are general, some specific to particular items
//

LIST OffOn = off, on
LIST SeenUnseen = unseen, seen

LIST GlassState = (none), steamed, steam_gone
LIST BedState = (made_up), covers_shifted, covers_off, stain_visible

//
// System: inventory
//

LIST Inventory = (none), cane, knife

=== function get(x)
    ~ Inventory += x

//
// System: positioning things
// Items can be put in and on places
//

LIST Supporters = on_desk, on_floor, on_bed, under_bed, held, with_joe

=== function move_to_supporter(ref item_state, new_supporter) ===
    ~ item_state -= LIST_ALL(Supporters)
    ~ item_state += new_supporter
```

(continues on next page)

(continued from previous page)

```
//
// System: Incremental knowledge.
// Each list is a chains of facts. Each fact supercedes the fact␣
↪before it.
//


LIST BedKnowledge = (none), neatly_made, crumpled_duvet, hastily_
↪remade, body_on_bed, murdered_in_bed, murdered_while_asleep
LIST KnifeKnowledge = (none), prints_on_knife, joe_seen_prints_on_
↪knife,joe_wants_better_prints, joe_got_better_prints
LIST WindowKnowledge = (none), steam_on_glass, fingerprints_on_glass,␣
↪fingerprints_on_glass_match_knife

VAR knowledgeState = ()

=== function learn(x) ===
    // learn this fact
    ~ knowledgeState += x

=== function learnt(x) ===
    // have you learnt this fact, or indeed a stronger one
    ~ return highest_state_for_set_of_state(x) >= x

=== function between(x, y) ===
    // are you between two ideas? Not necessarily in the same␣
↪knowledge tree.
    ~ return learnt(x) && not learnt(y)

=== function think(x) ===
    // is this your current "strongest" idea in this knowledge set?
    ~ return highest_state_for_set_of_state(x) == x

=== function highest_state_for_set_of_state(x) ===
    ~ return LIST_MAX(knowledgeState ^ LIST_ALL(x))

=== function did_learn(x) ===
    //  did you learn this particular fact?
    ~ return knowledgeState ? x

//
// Set up the scene
//
```

(continues on next page)

(continued from previous page)

```
VAR bedroomLightState = (off, on_desk)
VAR knifeState = (under_bed)

//
// Content
//

=== murder_scene ===
    The bedroom. This is where it happened. Now to look for clues.
- (top)
    { bedroomLightState ? seen:     <- seen_light  }
    <- compare_prints(-> top)

    *   (dobed) [The bed...]
        The bed was low to the ground, but not so low something might␣
↪not roll underneath. It was still neatly made.
        ~ learn(neatly_made)
        - - (bedhub)
        * *     [Lift the bedcover]
                I lifted back the bedcover. The duvet underneath was␣
↪crumpled.
                ~ learn(crumpled_duvet)
                ~ BedState = covers_shifted
        * *     (uncover) {learnt(crumpled_duvet)}
                [Remove the cover]
                Careful not to disturb anything beneath, I removed the␣
↪cover entirely. The duvet below was rumpled.
                Not the work of the maid, who was conscientious to a␣
↪point. Clearly this had been thrown on in a hurry.
                ~ learn(hastily_remade)
                ~ BedState = covers_off
        * *     (duvet) {BedState == covers_off} [ Pull back the duvet␣
↪]
                I pulled back the duvet. Beneath it was a sheet,␣
↪sticky with blood.
                ~ BedState = stain_visible
                ~ learn(body_on_bed)
                Either the body had been moved here before being␣
↪dragged to the floor - or this is was where the murder had taken␣
↪place.
        * *     {!(BedState ? made_up)} [ Remake the bed ]
                Carefully, I pulled the bedsheets back into place,␣
↪trying to make it seem undisturbed.
                ~ BedState = made_up
        * *     [Test the bed]
```

(continues on next page)

(continued from previous page)

```
                I pushed the bed with spread fingers. It creaked a␣
↪little, but not so much as to be obnoxious.
        * *     (darkunder) [Look under the bed]
                Lying down, I peered under the bed, but could make␣
↪nothing out.

        * *     {TURNS_SINCE(-> dobed) > 1} [Something else?]
                I took a step back from the bed and looked around.
                -> top
        - -     -> bedhub

    *   {darkunder && bedroomLightState ? on_floor &&␣
↪bedroomLightState ? on}
        [ Look under the bed ]
        I peered under the bed. Something glinted back at me.
        - - (reaching)
        * *     [ Reach for it ]
                I fished with one arm under the bed, but whatever it␣
↪was, it had been kicked far enough back that I couldn't get my␣
↪fingers on it.
                -> reaching
        * *     {Inventory ? cane} [Knock it with the cane]
                -> knock_with_cane

        * *     {reaching > 1 } [ Stand up ]
                I stood up once more, and brushed my coat down.
                -> top

    *   (knock_with_cane) {reaching && TURNS_SINCE(-> reaching) >= 4 &&
↪   Inventory ? cane } [Use the cane to reach under the bed ]
        Positioning the cane above the carpet, I gave the glinting␣
↪thing a sharp tap. It slid out from the under the foot of the bed.
        ~ move_to_supporter( knifeState, on_floor )
        * *     (standup) [Stand up]
                Satisfied, I stood up, and saw I had knocked free a␣
↪bloodied knife.
                -> top
        * *     [Look under the bed once more]
                Moving the cane aside, I looked under the bed once␣
↪more, but there was nothing more there.
                -> standup

    *   {knifeState ? on_floor} [Pick up the knife]
        Careful not to touch the handle, I lifted the blade from the␣
↪carpet.
```

(continues on next page)

(continued from previous page)

```
        ~ get(knife)

    *   {Inventory ? knife} [Look at the knife]
        The blood was dry enough. Dry enough to show up partial prints␣
↪on the hilt!
        ~ learn(prints_on_knife)

    *   [   The desk... ]
        I turned my attention to the desk. A lamp sat in one corner, a␣
↪neat, empty in-tray in the other. There was nothing else out.
        Leaning against the desk was a wooden cane.
        ~ bedroomLightState += seen
        - - (deskstate)
        * *     (pickup_cane) {Inventory !? cane}    [Pick up the cane␣
↪]
                ~ get(cane)
                I picked up the wooden cane. It was heavy, and␣
↪unmarked.

        * *    { bedroomLightState !? on } [Turn on the lamp]
                -> operate_lamp ->
        * *     [Look at the in-tray ]
                I regarded the in-tray, but there was nothing to be␣
↪seen. Either the victim's papers were taken, or his line of work had␣
↪seriously dried up. Or the in-tray was all for show.
        + +     (open)  {open < 3} [Open a drawer]
                I tried {a drawer at random|another drawer|a third␣
↪drawer}. {Locked|Also locked|Unsurprisingly, locked as well}.

        * *     {deskstate >= 2} [Something else?]
                I took a step away from the desk once more.
                -> top
        - -     -> deskstate

    *     {(Inventory ? cane) && TURNS_SINCE(-> deskstate) <= 2}␣
↪[Swoosh the cane]
        I was still holding the cane: I gave it an experimental swoosh.
↪ It was heavy indeed, though not heavy enough to be used as a␣
↪bludgeon.
        But it might have been useful in self-defence. Why hadn't the␣
↪victim reached for it? Knocked it over?

    *   [The window...]
        I went over to the window and peered out. A dismal view of the␣
↪little brook that ran down beside the house.
```

(continues on next page)

(continued from previous page)

```
        - - (window_opts)
        <- compare_prints(-> window_opts)
        * *     (downy) [Look down at the brook]
                { GlassState ? steamed:
                    Through the steamed glass I couldn't see the brook.
↪ -> see_prints_on_glass -> window_opts
                }
                I watched the little stream rush past for a while. The␣
↪house probably had damp but otherwise, it told me nothing.
        * *     (greasy) [Look at the glass]
                { GlassState ? steamed: -> downy }
                The glass in the window was greasy. No one had cleaned␣
↪it in a while, inside or out.
        * *     { GlassState ? steamed && not see_prints_on_glass &&␣
↪downy && greasy }
                [ Look at the steam ]
                A cold day outside. Natural my breath should steam. ->␣
↪see_prints_on_glass ->
        + +     {GlassState ? steam_gone} [ Breathe on the glass ]
                I breathed gently on the glass once more.
↪{learnt(fingerprints_on_glass): The fingerprints reappeared. }
                ~ GlassState = steamed

        + +     [Something else?]
                { window_opts < 2 || learnt(fingerprints_on_glass) ||␣
↪GlassState ? steamed:
                    I looked away from the dreary glass.
                    {GlassState ? steamed:
                        ~ GlassState = steam_gone
                        <> The steam from my breath faded.
                    }
                    -> top
                }
                I leant back from the glass. My breath had steamed up␣
↪the pane a little.
               ~ GlassState = steamed
        - -     -> window_opts



    *   {top >= 5} [Leave the room]
        I'd seen enough. I {bedroomLightState ? on:switched off the␣
↪lamp, then} turned and left the room.
        -> joe_in_hall
    -   -> top
```

(continues on next page)

(continued from previous page)

```
= see_prints_on_glass
    ~ learn(fingerprints_on_glass)
    {But I could see a few fingerprints, as though someone had leant␣
↪their palm against it.|The fingerprints were quite clear and well-
↪formed.} They faded as I watched.
    ~ GlassState = steam_gone
    ->->

= compare_prints (-> backto)
    *   {learnt(fingerprints_on_glass) && learnt(prints_on_knife) && !
↪learnt(fingerprints_on_glass_match_knife)} [Compare the prints on␣
↪the knife and the window ]
        Holding the bloodied knife near the window, I breathed to␣
↪bring out the prints once more, and compared them as best I could.
        Hardly scientific, but they seemed very similar - very␣
↪similiar indeed.
        ~ learn(fingerprints_on_glass_match_knife)
        -> backto

= operate_lamp
    I flicked the light switch.
    { bedroomLightState ? on:
        <> The bulb fell dark.
        ~ bedroomLightState += off
        ~ bedroomLightState -= on
    - else:
        { bedroomLightState ? on_floor: <> A little light spilled␣
↪under the bed.} { bedroomLightState ? on_desk : <> The light gleamed␣
↪on the polished tabletop. }
        ~ bedroomLightState -= off
        ~ bedroomLightState += on
    }
    ->->

= seen_light
    *   {!(bedroomLightState ? on)} [ Turn on lamp ]
        -> operate_lamp ->

    *   { !(bedroomLightState ? on_bed)  && BedState ? stain_visible }
        [ Move the light to the bed ]
        ~ move_to_supporter(bedroomLightState, on_bed)
        I moved the light over to the bloodstain and peered closely at␣
↪it. It had soaked deeply into the fibres of the cotton sheet.
        There was no doubt about it. This was where the blow had been␣
↪struck.
```

(continues on next page)

(continued from previous page)

```
        ~ learn(murdered_in_bed)

    *   { !(bedroomLightState ? on_desk) } {TURNS_SINCE(-> floorit) >=␣
↪2 }
        [ Move the light back to the desk ]
        ~ move_to_supporter(bedroomLightState, on_desk)
        I moved the light back to the desk, setting it down where it␣
↪had originally been.
    *   (floorit) { !(bedroomLightState ? on_floor) && darkunder }
        [Move the light to the floor ]
        ~ move_to_supporter(bedroomLightState, on_floor)
        I picked the light up and set it down on the floor.
    -   -> top

=== joe_in_hall
    My police contact, Joe, was waiting in the hall. 'So?' he demanded.
↪ 'Did you find anything interesting?'
- (found)
    *   {found == 1} 'Nothing.'
        He shrugged. 'Shame.'
        -> done
    *   { Inventory ? knife } 'I found the murder weapon.'
        'Good going!' Joe replied with a grin. 'We thought the␣
↪murderer had gotten rid of it. I'll bag that for you now.'
        ~ move_to_supporter(knifeState, with_joe)
    *   {learnt(prints_on_knife)} { knifeState ? with_joe }
        'There are prints on the blade[.'],' I told him.
        He regarded them carefully.
        'Hrm. Not very complete. It'll be hard to get a match from␣
↪these.'
        ~ learn(joe_seen_prints_on_knife)
    *   { learnt(fingerprints_on_glass_match_knife) && learnt(joe_seen_
↪prints_on_knife) }
        'They match a set of prints on the window, too.'
        'Anyone could have touched the window,' Joe replied␣
↪thoughtfully. 'But if they're more complete, they should help us get␣
↪a decent match!'
        ~ learn(joe_wants_better_prints)
    *   { between(body_on_bed, murdered_in_bed)}
        'The body was moved to the bed at some point[.'],' I told him.
↪'And then moved back to the floor.'
        'Why?'
        * *     'I don't know.'
                Joe nods. 'All right.'
        * *     'Perhaps to get something from the floor?'
```

(continues on next page)

(continued from previous page)

```
                'You wouldn't move a whole body for that.'
        * *     'Perhaps he was killed in bed.'
                'It's just speculation at this point,' Joe remarks.
    *   { learnt(murdered_in_bed) }
        'The victim was murdered in bed, and then the body was moved␣
↪to the floor.'
        'Why?'
        * *     'I don't know.'
                Joe nods. 'All right, then.'
        * *     'Perhaps the murderer wanted to mislead us.'
                'How so?'
            * * *   'They wanted us to think the victim was awake[.'],␣
↪I replied thoughtfully. 'That they were meeting their attacker,␣
↪rather than being stabbed in their sleep.'
            * * *   'They wanted us to think there was some kind of␣
↪struggle[.'],' I replied. 'That the victim wasn't simply stabbed in␣
↪their sleep.'
            - - -   'But if they were killed in bed, that's most␣
↪likely what happened. Stabbed, while sleeping.'
                    ~ learn(murdered_while_asleep)
        * *     'Perhaps the murderer hoped to clean up the scene.'
                'But they were disturbed? It's possible.'


    *   { found > 1} 'That's it.'
        'All right. It's a start,' Joe replied.
        -> done
    -   -> found
-   (done)
    {
    - between(joe_wants_better_prints, joe_got_better_prints):
        ~ learn(joe_got_better_prints)
        <>  He moved for the door.  'I'll get those prints from the␣
↪window now.'
    - learnt(joe_seen_prints_on_knife):
        <> 'I'll run those prints as best I can.'
    - else:
        <> 'Not much to go on.'
    }
    -> END
```

8) Summary

To summarise a difficult section, **ink**'s list construction provides:

Flags

- Each list entry is an event
- Use += to mark an event as having occurred
- Test using `?` and `!?`

Example:

```
LIST GameEvents = foundSword, openedCasket, metGorgon
{ GameEvents ? openedCasket }
{ GameEvents ? (foundSword, metGorgon) }
~ GameEvents += metGorgon
```

State machines

- Each list entry is a state
- Use = to set the state; ++ and −− to step forward or backward
- Test using ==, > etc

Example:

```
LIST PancakeState = ingredients_gathered, batter_mix, pan_hot,␣
↪pancakes_tossed, ready_to_eat
{ PancakeState == batter_mix }
{ PancakeState < ready_to_eat }
~ PancakeState++
```

Properties

- Each list is a different property, with values for the states that property can take (on or off, lit or unlit, etc)
- Change state by removing the old state, then adding in the new
- Test using `?` and `!?`

Example:

```
LIST OnOffState = on, off
LIST ChargeState = uncharged, charging, charged

VAR PhoneState = (off, uncharged)

```

(continues on next page)

(continued from previous page)

```
*   {PhoneState !? uncharged } [Plug in phone]
    ~ PhoneState -= LIST_ALL(ChargeState)
    ~ PhoneState += charging
    You plug the phone into charge.
*   { PhoneState ? (on, charged) } [ Call my mother ]
```

2.3.7 Unfold Studio extensions to Ink

Unfold Studio currently runs on Ink 0.8.2, with a few customizations.

Include

Unfold Studio handles includes a bit differently from Ink. For one thing, you can only include stories that are public or shared (even if they are your own), and stories are included by their ID, not a filename.

The rules for inclusion are also different. In regular Ink, defining the same knot or variable in two stories which are included together results in an error. In Unfold Studio, the one in the including story is kept and the other is ignored.

Styling

Coming soon, it will be possible to add images and styled text to stories.

2.4 Terms of Service

Terms of service and privacy policies are coming. For now, please know that Unfold Studio is a free, open-source academic research project, and any research based on it is covered by an IRB.

2.4.1 Privacy policy

Coming.

CHAPTER

THREE

TEACHING GUIDE

3.1 Pedagogy

Unfold Studio was developed through workshops with middle- and high-school students. These workshops also developed some teaching strategies that can make teaching with Unfold Studio more effective.

3.1.1 But I don't know Computer Science

Great! This is an opportunity to grow with your students–when you step out of the role of the expert in the room, it makes space for your students to take on leadership roles. While this can definitely feel uncomfortable, you may be pleasantly surprised to find that they (and you!) rise to the challenge.

Besides, even if you're not an expert computer scientist, you are an expert in whatever it is you teach. Focus on your subject area goals, and let the CS skills be something you and your students learn together. Many content-area applications of Unfold Studio are differentiated, so that a successful project can use just a bit of programming or can be very complex. (This is called "low floor-high ceiling.") Whether it's creating a simulation of an ecosystem, writing interactive historical fiction, or thinking about different paths through a legal argument, you can use Unfold Studio to teach your content area. The *Curriculum Unit* is an example of how computer science could be integrated into a sociology course.

3.1.2 Writer's Workshop

Writer's Workshop is a pedagogical strategy from English/Language Arts, in which the class is structured around students writing and sharing their own stories. It takes some work to build a classroom community where students eagerly come into class and settle into their writing, (especially if they are used to having discipline imposed by an authority figure) but the payoff can be so rewarding. When your students have the opportunity to tell their stories and know they'll be taken seriously by their teachers and their peers, they may be willing to bring more of their lives into the classroom, and work really hard to craft their stories. There are many resources available (TODO: CITE) for supporting Writer's Workshop in your classroom.

Writer's Workshop works particularly well in Unfold Studio, as everybody's published stories are available on the website and playing stories is an active process. Many students enjoy watching their peers play and replay their stories, and then discussing how they felt, and why they made certain choices.

3.1.3 Mini-lessons

Mini-lessons are short (20 minutes tops) focused lessons tailored to particular skills. They work particularly well when students are invited, not required, to attend. This puts a good check on you too–if nobody chooses to come to the mini-lesson, maybe it wasn't needed. Mini-lessons go hand-in-hand with Writer's Workshop; when students are in charge of their own time, they can decide whether it's more productive to work on their own or to go to the mini-lesson. (One student asked, "Is it ok for me to sit close to the mini-lesson and listen in, but mostly do my own work?")

We often offer mini-lessons on writing topics such as using dialogue, sensory details, irony, and character-development, as well as CS topics such as using variables, conditionals, state, and design patterns like a game with an inventory or creating characters who have variable qualities (level of trust, fatigue, exasperation, etc.). You might also offer mini-lessons on content-area goals. Here is a collection of *Mini-lessons* we have developed.

Once your students have gotten familiar with mini-lessons, you can ask them what topics would help them, or invite them to teach their own mini-lessons.

3.1.4 Collaboration and pair programming

Contrary to the stereotype of programming as a solitary activity, professional computer science is deeply collaborative: there is constant communication with clients and team-mates; colleagues critique each other's work in code reviews, and programs (along with documentation) build a shared understanding of a problem. It is common for open-source software projects to be co-written by hundreds of collaborators, whose forums and chatrooms buzz with arguments, suggestions, brainstorming, and support.

There are many ways you can support collaborative practice in your teaching as well, even if you're stuck using one of those computer labs with fixed rows of solitary workstations. Encourage your students to support each other, to move around, and to work together. One common pattern is pair-programming, in which two students sit at one computer, periodically rotating who is typing. This encourages more thoughtful programming.

Note: In the near future, Unfold Studio will support real-time collaobration in the same story. This will unlock many new collaborative possibilities.

3.2 Curriculum Unit

This five-week curriculum unit provides an example of how Unfold Studio can be integrated into a secondary English, Social Studies, or Computer Science course. You may want to use it whole, or mix and match lessons and resources to make it fit your discipline and teaching context. This unit is released free for noncommercial use. It was written by Chris Proctor, who would love to receive your questions and feedback. (see *Contact*)

Note: This curriculum unit follows the backwards-planning format suggested in *Understanding by Design* [WM00]. The jumping-off point is an essential question, something worth thinking about for a long time and which leads to important ideas. Pursuing the essential question provides opportunities for particular learning objectives and assessments to measure growth. Once this planning is finished, we are ready to plan out the lesson day-by-day.

3.2.1 Introduction

Who can we be? The social worlds we live in provide both resources and limitations for the kinds of identities we can write for ourselves. We all know how great it feels to be in a space where you feel safe and free. We are also familiar with how some spaces keep us from being the people we would like to be. Maybe it's because your parents have a fixed idea about who you are, and impose that on you. Maybe it's because you're not around the people who know you best, and with whom you have shared memories. Maybe it's because you're in a culture that attaches expectations to your gender, race, or social position.

This unit invites students to explore the relationship between language, identity, and literacy spaces. Literacy spaces are communities of meaning-making which have their own histories with ideas and ways of understanding media. Your family is a literacy space–its members are known in particular ways, you know what gets talked about (and what doesn't) at the dinner table, and everyone knows what to laugh about and what to take seriously. Other examples of literacy spaces are school classrooms and Facebook.

In a typical US middle or high school, these questions would often be explored in an English, Language Arts, or Social Studies class. They might also be right at home in a Computer Science class. We are still developing an idea of what secondary Computer Science should look like, but these questions are of central importance to human-computer interaction, modeling social phenomena, and computational linguistics. This unit is an example of how Unfold Studio can support interdisciplinary literacies, using computational tools to explore humanistic questions.

3.2.2 Objectives

1. Use the concepts of *identity*, *model of personhood*, *register*, and *literacy space* to analyze real-world social experiences.

2. Write interactive stories which critically engage with real-world social experiences.
3. Plan interactive stories using planning and prewriting strategies such as freewriting and using *graphs* to plan story flow.
4. Use *flow control* and *state* to write interactive stories. Specifically, make use of:
    - Branches (see *5) Branching The Flow*)
    - Variables (see *Part 3: Variables and Logic*)
    - Advanced: Tunnels and Threads (see *Part 4: Advanced Flow Control*)
    - Advanced: Lists (see *Part 5: Advanced State Tracking*)

3.2.3 Assessment

Three assessments will allow students to demonstrate mastery of these objectives:

- The *Writing Process Assesssment* assesses students' prewriting and planning strategies (Objective 3). Students submit prewriting documents, a story draft, and a reflection explaining the prewriting documents.
- The *Story Portfolio* assesses students' ability to analyze and critically engage with real-world social experiences (Objectives 1 & 2). Students submit several stories and a reflection explaining their intent.
- The *Broken Story* assesses students' ability to use programming concepts (Objective 4). Students are given a broken story and asked to fix it.

The writer's workshop structure provides plenty of opportunities for informal formative assessment. The unit also contains several structured opportunities to check how students are doing.

3.2.4 Daily Lesson Plans

Table 1: Unit Calendar, assuming 50-minute class periods

Day 1: Introduction	*Day 2: Map of childhood*	*Day 3: Lesson*	*Day 4: Lesson*	*Day 5: Lesson*
Day 6: Lesson	*Day 7: Lesson*	*Day 8: Lesson*	*Day 9: Lesson*	*Day 10: Lesson*
Day 11: Lesson	*Day 12: Lesson*	*Day 13: Lesson*	*Day 14: Lesson*	*Day 15: Lesson*
Day 16: Lesson	*Day 17: Lesson*	*Day 18: Lesson*	*Day 19: Lesson*	*Day 20: Lesson*
Day 21: Lesson	*Day 22: Lesson*	*Day 23: Lesson*	*Day 24: Lesson*	*Day 25: Lesson*

Todo: Write daily lesson plans

Day 1: Introduction

Introductory discussion: Who can we be? Quick free-write on particular questions; class discussion; free-writing. Create accounts on Unfold Studio (unless private installation); basics of syntax; first story

Day 2: Map of childhood

Map of childhood activity.

Day 3: Lesson

Map of childhood activity: finish up and share. Discuss interesting effects people created, different feelings in stories.

Day 4: Lesson

Prewriting and perspective story. Implement the story. Working in pairs.

Day 5: Lesson

Finish perspective story. Share, appreciate as a class.

Day 6: Lesson

Lecture and discussion on big concepts: models of personhood. Got another text? Incorporate it. Choose one of three writing prompts (from slides); freewrite on this and submit to teacher as a formative assessment.

Day 7: Lesson

Return writing; implement story. Set the stage: a little longer to write, building up. Will take time to discuss. Mini-lesson on conditionals (or let people figure it out on their own)

Day 8: Lesson

Writer's workshop day on dialogue story. Mini-lesson on state.

Day 9: Lesson

Finish up dialogue stories, share, discuss.

Day 10: Lesson

Literature circles: read a story, practice discussing it. Whole-class discussion: What was interesting? dialogic interactions that need more attention. Introduce assessments, milestones, writer's workshop working structure.

Day 11: Lesson

Writer's workshop. Mini-lesson on pre-writing. Exit ticket: what do you need mini-lessons on?

Day 12: Lesson

Writer's workshop Mini-lesson on dialogue

Day 13: Lesson

Writer's workshop Mini-lesson on inventory

Day 14: Lesson

Writer's workshop. Mini-lesson on theory of mind Assignment: literature circles meet, decide on a class-written story to read for tomorrow.

Day 15: Lesson

First story due. Literature circles meet, discuss class-written story.

Day 16: Lesson

Writer's workshop

Day 17: Lesson

Writer’s workshop

Day 18: Lesson

Writer’s workshop

Day 19: Lesson

Writer’s workshop

Day 20: Lesson

Writing process assessment due

Day 21: Lesson

Writer’s workshop

Day 22: Lesson

Writer’s workshop

Day 23: Lesson

Broken story assessment

Day 24: Lesson

Finishing up portfolios

Day 25: Lesson

Closing discussion, reflective writing

3.3 Teaching Resources

These resources may be helpful in teaching with Unfold Studio. They can be used as part of a coherent *Curriculum Unit*, or you can mix and match them to fit your needs. Unless otherwise noted, all were written by Chris Proctor, and are released free for noncommercial use.

3.3.1 Assessments

Writing Process Assesssment

Todo: Describe the portfolio and write a rubric.

This is about the process of coming up with a story idea, prewriting, developing, getting feedback, revising, reflecting.

Story Portfolio

Todo: Describe the portfolio and write a rubric.

This is mostly about using stories as social analytic artifacts. Using computational affordances.

Broken Story

Students are given a broken interactive story and asked to fix it. Here is an example you may use as-is or adapt a version to fit your teaching context and the skills you want to assess.

This strategy for assessing computational thinking was developed by [WDCK12]. Creating a standardized assessment of computational thinking is tricky because a student's ability to demonstrate mastery depends on her mastery of the programming language being used. The Broken Story assessment is particularly attractive for Unfold Studio because the assessment is much more like the practice of writing interactive stories than something like a multiple-choice test. It is also possible to automatically assess Broken Stories, though this is not yet available.

Rubric

Table 2: Broken Story Rubric

Skill	Evidence
Syntax	• Story compiles with no errors
Choices match the graph	• Add a choice to go from `enter_peets` to `use_wifi`.
Sticky choices	• Make choice from `use_wifi` to `read_news` sticky (+) • Make choice from `use_wifi` to `minecraft` sticky (+) • Make choice from `read_news` to `minecraft` sticky (+) • Make choice from `minecraft` to `read_news` sticky (+)
Conditional choices	• `read_news` to `buy_tea` should have `{not buy_tea}` condition • `read_news` to `leave` should have `{buy_tea}` condition
Conditional text	• `leave` should have `{minecraft: text | other text}`

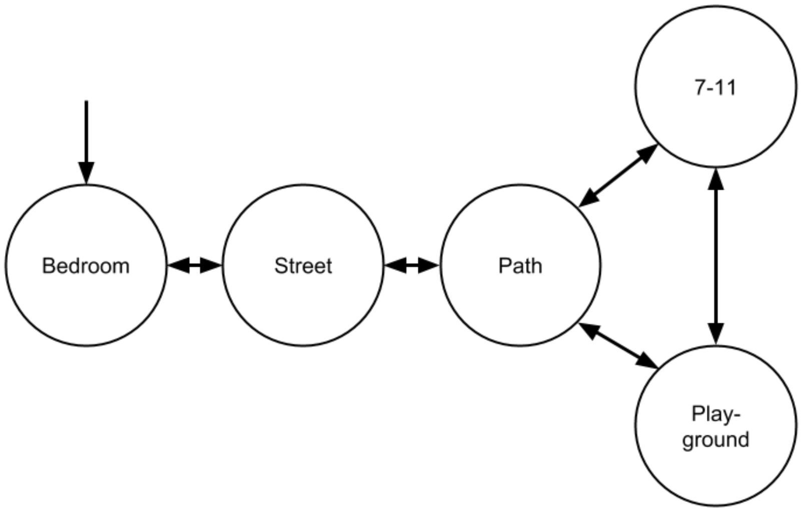
3.3.2 Story Prompts

These prompts can be helpful in introducing new skills or concepts, or as prewriting exercises.

A map of your childhood

Note: This story prompt is a great first introduction to ink. It demonstrates some of the core language features and invites authors to explore using a simple structure.

Think of about five places you remember well from your childhood. Draw a graph showing how the places are connected to each other, like the one below. Choose one place where the player should start and add an arrow pointing to that place. Then take some time to freewrite descriptions of each place, as well as how you get from one place to another. For now, there will be no conditions on moving between connected places–it's just a map the player can wander around.



Now write this as an interactive story. Here is an example. Hold off on adding lots of description until you're sure it works.

```
It's 11pm on a Friday night, sometime in ninth grade. Two friends are␣
↪sleeping over.
-> bedroom

=== bedroom ===
You are in your bedroom.
+ Go outside
    -> street

=== street ===
You are on a quiet, moonlit street.
+ Go back inside.
    -> bedroom
+ Walk to the end of the street.
    -> path

=== path ===
You are on a dark path.
+ Go back onto the street.
    -> street
+ Walk to 7-11.
-> seven_eleven
+ Walk to the playground.
    -> playground
```

(continues on next page)

(continued from previous page)

```

=== seven_eleven ===
You walk to 7-11. The garishly-bright interior feels incongruous
with the peaceful night.
+ Walk to the playground.
    -> playground
+ Walk back to the path.
    -> path

=== playground ===
You sit on the swings of a deserted playground.
+ Walk to 7-11.
    -> seven_eleven
+ Walk back to the path.
    -> path
```

Now think of two or three objects which might be found in this world, particularly objects that have some meaning for you. We are going to extend the world-map so that the player can collect these objects. Then we'll have the world change depending on which objects the player has collected. To keep it simple, the model story will only add one object: a Slurpee. We'll use a simple boolean (true/false) variable to keep track of whether we have a Slurpee. Note that we need to initialize the variable (line 1) before we can modify it (line 38).

```
1  VAR has_slurpee = false
2
3  It's 11pm on a Friday night, sometime in ninth grade. Two friends are sleeping over.
4  -> bedroom
5
6  === bedroom ===
7  You are in your bedroom.
8  + Go outside
9      -> street
10
11 === street ===
12 You are on a quiet, moonlit street.
13 + Go back inside.
14     -> bedroom
15 + Walk to the end of the street.
16     -> path
17
18 === path ===
19 You are on a dark path.
20 + Go back onto the street.
21     -> street
```

(continues on next page)

(continued from previous page)

```
+ Walk to 7-11.
-> seven_eleven
+ Walk to the playground.
    -> playground

=== seven_eleven ===
You walk to 7-11. The garishly-bright interior feels incongruous
with the peaceful night.
+ Buy a Slurpee.
   -> buy_slurpee
+ Walk to the playground.
    -> playground
+ Walk back to the path.
    -> path

=== buy_slurpee ===
~ has_slurpee = true
Your sleepy eyes are mesmerized by the spinning slush in the Slurpee␣
↪machine.
You slowly fill the cup with cherry slush, then switch to root beer
all the way to the top of the rounded lid.
+ Walk to the playground.
    -> playground
+ Walk back to the path.
    -> path

=== playground ===
You sit on the swings of a deserted playground.
+ Walk to 7-11.
    -> seven_eleven
+ Walk back to the path.
    -> path
```

Now the player can get a Slurpee, but it has no effect on the rest of the story. Let's make the story react. This is an opportunity to craft an experience for your readers: when they take an action, even something seemingly insignificant like choosing to buy a Slurpee, they become invested in your world. This can be a chance to let them feel an emotion, have a sensory experience, or experience life in someone else's shoes.

```
VAR has_slurpee = false

It's 11pm on a Friday night, sometime in ninth grade. Two friends are␣
↪sleeping over.
-> bedroom

```

(continues on next page)

(continued from previous page)

```
=== bedroom ===
You are in your bedroom.
+ Go outside
    -> street

=== street ===
You are on a quiet, moonlit street.
+ Go back inside.
    -> bedroom
+ Walk to the end of the street.
    -> path

=== path ===
You are on a dark path.
+ Go back onto the street.
    -> street
+ Walk to 7-11.
-> seven_eleven
+ Walk to the playground.
    -> playground

=== seven_eleven ===
You walk to 7-11. The garishly-bright interior feels incongruous
with the peaceful night.
{has_slurpee:
    If you go back in there with your Slurpee, the cashier might think you
    haven't paid for it. Better not to take the risk. You take an idle sip
    and wander off into the night.
    -> path
}

+ Buy a Slurpee.
   -> buy_slurpee
+ Walk to the playground.
    -> playground
+ Walk back to the path.
    -> path

=== buy_slurpee ===
~ has_slurpee = true
Your sleepy eyes are mesmerized by the spinning slush in the Slurpee machine.
You slowly fill the cup with cherry slush, then switch to root beer
```

(continues on next page)

(continued from previous page)

```
all the way to the top of the rounded lid.
+ Walk to the playground.
    -> playground
+ Walk back to the path.
    -> path

=== playground ===
{has_slurpee:
    You sit on the swings of a deserted playground, gently rocking as you sip
    on your Slurpee. You remember other nights here with friends, arguing
    about philsophy until the fringe of dawn appeared in the sky.
- else:
    The playground is deserted, completely still in the moonlight. You feel
    the night mist settling over you and shiver.
}
+ Walk to 7-11.
    -> seven_eleven
+ Walk back to the path.
    -> path
```

Collaborative world-building

This is about discovering the possibilities in working together to create a story.

Architecture: there should be one central story which stubs out all the locations and intiializes all the variables. Make this public so everyone can update it.

People might want to grow this further; they could create libraries of descriptions or functions to re-use. You might be able to create various characters who could be put in conversation with one another!

Argumentation

Many disciplines have an essentially dialogic argumentation style (Elbow's You Say, I Say). You could implement this in stories. Particularly valuable for seeing how people argue through stories.

Literary analysis

Multiple levels of meaning. I'm currently working on an example where the main character reads and re-reads a sonnet, layering in more possibilities each time.

Personalizing political, economic, and historical forces

Write a first-person story in which the main character interacts with broad-scale political, economic, or historical forces. The consequences following a player's choices should illustrate how the forces work, and should be supported by evidence. As part of the research process, interview at least two people who have experienced similar situations. Part of each interview should include playing and discussing your draft of the story.

For example, you might read this article about cash bail in America and then write an interactive story playing out the various choices someone might make after being arrested and unable to make bail. You might then interview someone who has been arrested, someone whose family member has been arrested, a police officer, an employer, etc.

3.3.3 Mini-lessons

Inventory

Todo: Add inventory mini-lesson

3.4 Workshops

We occasionally host workshops on Unfold Studio at academic conferences, public events, and professional development.

3.4.1 Mozfest 2017

In October 2017, Chris Proctor and Antero Garcia shared Unfold Studio at "Worldbuilding for Safe, Secure, and Private Futures: Producing Internet-Related Roleplaying Games and Interactive Fiction" at MozFest 2017 in London. Here are the [slides](https://docs.google.com/presentation/d/15x6sa46lWAgpgMR9-9HpfYogJdIQiHCTsD9ZVV0BG4s/edit?usp=sharing_eil&ts=59ee56ec)

3.4.2 ICLS 2018

In June 2018, Chris discussed Unfold Studio in a workshop on supporting and analyzing collaborative writing at the International Conference of the Learning Sciences in London, UK. Here are the [slides](http://chrisproctor.net/slides/2018-icls-workshop-presentation-interactive-storytelling.html)

3.4.3 CSTA 2018

In July 2018, Chris led a workshop on Unfold Studio at the Computer Science Teachers' Association annual conference in Omaha, Nebraska. Here are the [slides](http://chrisproctor.net/slides/2018-csta-workshop-interactive-storytelling.html).

3.4.4 Philly Celebration of Writing and Literacy

In October 2018, Chris led a workshop on Unfold Studio at the University of Pennsylvania with teachers from across Philadelphia.

3.5 Support

If you have questions about teaching with Unfold Studio or using it for other purposes, please get in touch with Chris Proctor (*Contact*), the creator and lead researcher on the project. He's happy to answer questions, or just brainstorm some ideas together.

The best place for technical questions, bug reports, and requested features is Unfold Studio's GitHub site, but you're welcome to send them by email if that is more comfortable.

3.5.1 Private installations

Many teachers choose to use the free public version of Unfold Studio and value the authentic audience it provides. However, this might not be the right fit for all schools. Private installations are also available, where access is restricted to logged-in members. Private installations also include the following features:

- User administration (LMS integration available)
- Teachers have access to a dashboard where they can view and administer student stories.
- Students can privately turn in stories to teachers

3.5.2 Professional development

Chris, an award-winning teacher and experienced provider of professional development, is available for customized PD with Unfold Studio on a range of scales. Unfold Studio can be a great introduction to Computer Science for students and teachers alike!

CHAPTER

FOUR

RESEARCH

Unfold Studio began as a research project during Chris Proctor's PhD at Stanford's Grauduate School of Education. As a former high school English teacher and a middle-school CS teacher, Chris was interested in how

Interactions between texts and identity and culture are important in English/Language Arts, but are largely missing from the conversation in computer science pedagogy.

4.1 Designing Unfold Studio

Todo: Participatory design research. . .

4.2 Learning with Unfold Studio

Todo: Summary of research questions and findings. Links to papers.

4.3 Become a research partner

We are actively conducting research with Unfold Studio, and are looking for teachers and schools to partner with. This could include grant-funded professional development and researcher support creating teaching materials. Whether you have particular research ideas, a unique teaching situation, or are just interested in exploring research possibilities, we would love to hear from you. Please see *Contact* for contact information.

4.4 References

CHAPTER

FIVE

CONTACT

Unfold Studio and this documentation was created by Chris Proctor, a PhD candidate in Learning Sciences and Technology Design at Stanford's Graduate School of Education. Chris is a member of the Transformative Learning Technologies Lab, led by his advisor Paulo Blikstein. Please feel free to contact Chris at cproctor@stanford.edu.

5.1 Citing Unfold Studio

5.1.1 The paper

Proctor, C. & Blikstein, P. (2018). *Unfold.studio: Suporting critical literacies of text & code.* Manuscript submitted for publication.

```
@unpublished{proctor2018,
    author = "Proctor, C. and Blikstein, P.",
    year   = 2018,
    title  = "Unfold.studio: Suporting critical literacies of text &␣
↪code.",
    notes  = "Manuscript in submission."
    url    = "http://chrisproctor.net/text-and-code.html"
}
```

5.1.2 The software

Proctor, C. (2018). Unfold Studio [Computer software]. Retrieved from https://github.com/cproctor/unfold_studio

```
@misc{
    author = "Proctor, C.",
    year   = 2018,
    title  = "Unfold Studio",
    url    = "https://unfold.studio",
    howpublished = {\url{https://github.com/cproctor/unfold_studio}},
}
```

5.1.3 The documentation (including teaching guide)

Proctor, C. (2018). Unfold Studio Documentation and Teaching Guide. Retrieved from http://docs.unfold.studio

```
@manual{proctor2018doc,
    author = "Proctor, C.",
    year   = 2018,
    title  = "Unfold Studio Documentation and Teaching Guide",
    url    = "http://docs.unfold.studio"
}
```

BIBLIOGRAPHY

[WDCK12] Linda Werner, Jill Denner, Shannon Campe, and Damon Chizuru Kawamoto. The fairy performance assessment: measuring computational thinking in middle school. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, 215–220. ACM, 2012.

[WM00] Grant Wiggins and Jay McTighe. *Understanding by Design*. Unknown, 2000.