Inform 7

Programmer's Manual

Contents

Ron Newcomb

February 6, 2011
build 6G60

pscion@yahoo.com

Inform 7: In a Nutshell

Inform 7 is a domain-specific language intended for authoring interactive fiction in the vein of Zork and Colossal Cave. Just as the videogame industry in general strives for better story in its products, Inform 7 strives to appeal to creators who are not programmers, and certainly do not have a collegiate background in computer programming with all of the vocabulary and metaphors that that entails. In this way, Inform 7 shares aims with BASIC, COBOL, and Applescript: a readable language for the intelligent layperson. But Inform takes readability much further.

This chapter compares Inform with other programming languages to give the seasoned programmer a sense of place. It can be skipped if so desired, though the rule breakdown at the end may be useful to return to later.

Inform does not concern itself about optimizing compilation speed at the expense of syntax, or ensuring scalability and security at the expense of complexity. Several design choices seem unusual or even a bad idea at first blush, but they are based on several years' worth of experience in creating interactive fiction. The first two of these choices will be much on your mind while learning. The other two will eventually befriend you.

First: identifiers may have spaces. Avoiding the underscore-or-capitalize contention, in Inform 7 one could name a variable "the currently owned car". Likewise for all other constructs in the language, including functions: **"mull over** (idea - an object) **in my** (spot - a room)**"**. Extraordinarily readable code results when constructs are named appropriately: objects and value-returning functions after noun phrases, actions and activities after participial phrases, non-value-returning functions after imperative sentences, etc. Articles *a*, *an*, and *the* are almost always ignored.

And second: rulebooks. Inform 7 is a rulebook-based language. A rulebook is a container for rules, but these are not the production rules from Prolog. Rather, they are unnamed imperative functions that self-invoke based on global variables, game-state, and most importantly, on whether the containing rulebook is executing. From the outside, a rulebook acts -- and is invoked -- like a function: give it a task and it will solve it. The difference is how it solves it. It may execute every applicable rule within itself, or may only execute the most specific applicable rule, your choice. The entirety of the Inform 7 system is a sequence of rulebooks in a proscribed order, with the oft-ignored background ones calling the numerous foreground rulebooks, and the foreground ones, initially empty, containing dozens of author-written rules.

Third are adjectives, especially as used by set-descriptions. An adjective is a boolean, but when combined with the class or construct to which they belong, they describe a set of instantiations. Consider this common loop construct, keywords being in bold: **"repeat with** antagonist **running through** unhappy resourceful people**"**. The index variable will be "antagonist". Person is a class, while unhappy and resourceful are boolean values for two different properties or methods. The loop will set antagonist to each person in turn whose adjectives are set in the indicated way. Inform recognizes plural words as synonymous.

Finally are relations. Object-oriented programming is built on two asymmetric relations, is-a and has-a. Inform allows creation of binary relations between almost any two constructs in the language, including text strings. Like adjectives, relations are either-or: either the friendship relation holds between two person instantiations or it doesn't. Moreover, many constructs can check a relation, the repeat loop being the simplest: "**repeat with** ally **running through** resourceful people which are friends with the player".

Of all programming languages, Inform 7 currently most resembles natural language. Knuth's ideas of literate programming reinforce this, as evidenced by the official website, Inform7.com. Terminology useful in describing Inform 7 follow.

Domain-specific. It compiles to a virtual machine not used outside of the interactive fiction community, and rules are restricted to three parameters: the highest arity a verb of English may have.

Event-based. Events are called "actions" because the game player's commands trigger protagonist actions in the simulated, turn-based gameworld.

Object-based. Single-inheritance and polymorphism are supported, but abstraction, modularity, and namespaces are poor to absent. Inform 7 is a white-box development language by design. Scalability and security are of no concern.

Macro language. Technically, Inform 7 is a macro language for Inform 6, a weakly-typed multiple-inheritance traditional programming language. All Inform 7 code becomes Inform 6 code before Inform 6 compiles to the virtual machine's assembly. The dev team takes care to hide this complication from novice users, but inline Inform 6 code, as well as inline VM assembly, can be used to powerful effect.

Strongly typed. You'll swear you're in Pascal again, but the type system supports generic types and type variables. When all else fails, inline Inform 6 code will circumvent.

Statically allocated. Although extensions can be installed to circumvent this, and text modified at runtime skirts it, Inform's motto is "one pointer, one block". This greatly simplifies things for the target audience even if all the deep copying slows naïve code.

Pronounceable, Type-able. Perhaps voice-recognition input methods for coding will one day become the norm. Until then, you can transcribe Inform code over your friend's telephone. Additionally, the Dvorak keyboard layout optimizes the placement of letters, which Inform makes almost exclusive use of. Dvorak doesn't do much for punctuation, which traditional languages favor.

Eschews One True Construct. Some languages can say "everything is an object" or "everything is a list". Inform does not have One True Construct, though the powerful type system can bring objects, enums, rules, etc. under the same umbrella for the same effect as having a common parent class. Theoretically, the lack of a unifying construct eases learning as each construct can be learned independent of anything else, and constructs implement themselves however makes sense.

Easy To Read, Hard To Write. Empirically, Inform has proven to be easy to read but hard to

write. The natural language appearance can be a will-o-wisp at times, luring the author from the path of clear grammar, as there are many phrasings that Inform "ought" to recognize, but does not. Conversely, creating uncommented code beautiful enough to be worth publishing is a viable sport. Such code can deliver its meaning even to those who aren't programmers.

Semantic Concision. Inform boasts some unique semantic conciseness on top of its extreme readability, which will hopefully be emulated in programming languages to come. For example, this is a single rule from a particular game.

> Instead of a suspicious person (called the suspect) burning something which is evidence against the suspect when the number of people in the location is at least two, try the suspect going a random valid direction.

That rule breaks down like this.

"**Instead**" is the rulebook which owns the rule. The rulebook's name comes first, and there's precisely one of them.
"**of**" is one of the many ignored prepositions allowed in that place, for readability.
"**a**" is an article, and ignored.
"**suspicious**" is an adjective on class Person.
"**person**" is the class of the first parameter to the burning action.
"**(called the suspect)**" names the first parameter so we can reference it elsewhere in the rule.
"**burning**" is the action, the event, usually triggered by the player entering "BURN EVIDENCE". It takes two parameters, of types Person (the actor) and Thing.
"**something**" is synonymous with "a thing" just as "someone" is synonymous with "a person". It is class Thing.
"**which**" flags that we're using a relation to narrow down what Things may trigger the rule.
"**is evidence against**" is a many-to-many relation, thing-to-person. Contrast the number-to-number “is less than”.
"**the suspect**" is to what we're relating, which in this case is the first parameter.
"**when**" is synonymous with "if", but is used in rule preambles while "if" is used in imperative code blocks.
"**the number of**" is a function that will count the members in the following set-description.
"**people**" is, again, a name for class Person.
"**in the location**" is a shortcut phrase for a person-to-room relation wherein the person is always the player.
"**is at least**" means equal to or greater than. If the suspect is alone, our rule won't apply.
"**two**" can be written as 2, but Inform is unusual in that digits tend to stick out like sore thumbs.
"**,**" the comma ends the rule preamble and begins the imperative code block. Frequently a colon must go here.
"**try**" will initiate a new action, and the whole sequence of rulebooks that that entails. In this case, the Going action will be called.
"**the suspect**" will apparently try a new tack in destroying the self-incriminating evidence he's holding, because he's not alone.
"**going**" takes two parameters, of types Person and Direction.
"**a random**" chooses one member out of a set-description. The following two words define this particular set of instantiations.
"**valid**" is an adjective on class Direction. It is rare the room that has exits in every direction!

"**direction**" is a class. The twelve standard instantiations are the eight compass points plus Up, In, Down, and Out.
"**.**" the period ends the rule. So would a blank line.

Rule preamble constructs like *someone hungry eating something edible* are not action invocations -- that would be *try Mario eating a mushroom* -- and not action definitions -- multiple rulebooks do that -- but rather, they are very much like a regex, applied to the state of the whole work rather than to a piece of text.

Excepting the automatically included Title and Author line on the first line of the source, the Hello World program is this.

> My apartment is a room. When play begins, say “Hello world.”

Every program must have at least one instantiation of class Room, which is where the virtual action takes place. (So the smallest program that will compile is: *Foo is a room.*) Comments belong in square brackets, and the IDE will color them green.

A final note: In this guide as well as in Inform's official documentation, "the parser" always refers to the runtime parser, our player's simplistic VERB NOUN PREPOSITION NOUN parser. It does not mean that part of the Inform 7 compiler itself.

The Firehose

"Information is gushing toward your brain like a firehose aimed at a teacup."
– Scott Adams, cartoonist, on the complexities of modern life (1996)

Because Inform 7 source is not cryptic, I don't feel we need much exposition before showing source code. It is fairly easy to see what any natural language-inspired programming language is trying to accomplish simply by reading source. The difficulty with learning Inform 7, besides the whole rulebook thing, will be trying to see the grammar and types behind the pretty façade so you can actually write it. So in this chapter, The Firehose, we go on a whirlwind tour of the language reading code samples and mapping common constructs into Inform. For brevity, the very first automatically-added line of the source which has the work's title and author won't be shown in this guide.

```
The apartment is a room. "Behold Bob's apartment. That smell is coming from the pile of dishes in the sink."

The apartment complex's lobby is south of the apartment. "Rows of mailboxes are set into the lobby wall. Box 114 is Bob's."

Mr Bob Dobalena is a man in the apartment. "Bob sports a well-loved Transformers t-shirt with acid-washed jeans."

T-shirt, jeans, and a pair of shoes are wearable things. Bob is wearing the shoes, the T-shirt, and the jeans.

Instead of examining the player, say "You've come to visit your old college friend, Bob, and are standing in [the location]."
```

We have two Room instances (*apartment* and *apartment complex's lobby*), one Man instance, and three Thing instances (*T-shirt*, *jeans*, *pair of shoes*) with their boolean *wearable* property set to true. The quoted text that's just floating by itself out there becomes the *initial appearance* property of Thing and its subclasses, or for anything else, the *description* property (not to be confused with the set-description type.) There's a few relations in there as well. The mapping-south relation ties the two rooms together, and must exist for *going* events on the obvious *direction* instance. By default, the mapping-north relation is also sensibly set. The wearing relation is thrice asserted to hold between Bob and each article of clothing. Finally, the *instead* rule changes the behavior of the Examine event, our player's command EXAMINE ME, which typically prints "As good looking as ever."

The source accomplishes a lot with little writing. One sentence instantiated three Things – the three pieces of clothing "are ... things" – and set a boolean property on them at the same time – all three pieces are "wearable things". The next sentence simultaneously set three relations, between Bob and each article of clothing. Object names can be abbreviated (*Bob*, *shoes*). The text that's printed in response to examining fills in the name of whichever room the player is standing in.

Next we'll show variable declarations of every major type. For now, know that *is* and *are* are perfectly synonymous and the articles *a*, *an*, and *the* are simply stripped from the source.

X **is** *a number* **that varies**. Y **is** *a number* **variable**.
Deadline **is** *a time* **that varies**.
An excuse **is** *some text* **that varies**.
My favorite toy **is** *a thing* **that varies**.
The current manager **is** *a person* **that varies**.
The ocean currents **are** *a direction* **that varies**.
The best spot **is** *a room* **that varies**.
The light switch's boolean **is** *a truth state* **that varies**.
The guru's answers **are** *a table name* **that varies**.
My secret plans **are** *a rulebook* **that varies.**
What worked last time **is** *a rule* **that varies.**
My regex target **is** *some indexed text* **that varies.**
An abeyance **is** *a stored action* **that varies.**

And now some examples of supplying initial values. *Usually* defines a default value that can be overridden by a specific *is*. *Usually* is very useful for creating class properties with a default value, which specific instances can override at compile-time. Same goes for globals.

X **is** usually 2. Y **is** 5.
Deadline **is** usually 4:30 pm.
The excuse **is** usually "I don't know"
My favorite toy **is** usually the red Ferrari.
The current manager **is** Bob.
The light switch's boolean **is** usually true.
The guru's answers **are** usually the table of deep answers.

Learning to write rules for the various player-generated events (*actions*) and library events (*activities*) is core. The rule's preamble is much like a regex on the state of the game, while the rule body is simply imperative code. All rules belong to a rulebook, and the rulebooks a game author usually writes action rules for are, in order of execution, *before*, *instead*, *check*, *carry out*, *after*, *report*, and *every turn*. Some of the usual player actions are: *looking*, *examining*, *taking*, *dropping*, *taking inventory*, *going*, *opening*, *closing*, *inserting it into*, *putting it on*, *removing it from*, plus a few system-level ("out of world") actions such as *saving the game*, *restoring the game*, and *quitting the game*. By default, an *instead* or *after* rule will cause execution to hop immediately to *every turn*, and a *check* rule is expected to do the same though by default it does not. Out of world actions only use the *check*, *carry out*, and *report* rules, and sometimes not even all of those. For example, once *quitting the game* is carried out, the software is no longer running to report anything.

It is easier to show all parts of a full rule preamble than explain each part in turn, rather than starting with simpler preambles and working one's way up. Ready your teacup.

A check rule for an actor inserting my favorite toy into the jeans when the time of day is before the deadline:
    say "[The current manager] would fire [the actor] for stuffing [my favorite toy] down there!" instead.

Syntax requires the rulebook name first, excepting articles like *a* which are practically whitespace. Here we use *check*. Rulebook names can be one word or many, and tend to be or start with either an imperative verb (check, carry out, report) or a subordinating conjunction (before, after, instead). That isn't required, but it reads better because actions are always a present participle (the -ing form of a verb). Rulebooks which don't typically hinge on an action, such as *every turn*, are named to stand alone, or mesh well with *when* or *while* which usually follows their name.

Next can be some optional words: *rule*, *for, of*. They do nothing except improve the flow of reading. Here we used both *rule* and *for* only for the sake of an example. *Instead* almost always uses *of*, but normally, we would have begun *Check an actor...*

Anyway, after the rulebook name is, optionally, the performer. Here we used *an actor* so the rule applies to all characters, not just the player. (The *an* is one of the few times when a particular article is required.) Other options are 2) omit it entirely, so the rule only applies to the player; 3) a specific named character such as Bob, provided Bob is not the player; 4) a set-description, such as *a resourceful person*; or 5) *someone,* which means anyone except the player. The imperative rule body references what matched with the variable *the actor*.

Next is the action. The action's full name is *inserting it into* but only the part before the *it* goes here. So, *inserting*. For the *before*, *instead*, and *after* rulebooks our options also include *doing something* which means any action, or a list of actions such as *looking, examining, or taking*, or variations on *doing something except looking, inserting, or examining*. For the exception list, we can check the parameters as well, but the arity of the check must be equal: *doing something except inserting or examining someone* would be rejected. But *doing something except inserting someone or examining someone* would be fine, as they both check the same arity.

Next is the noun fed to the action, such as whatever is currently in *my favorite toy*. We can use a set-description here as well. The variable *the noun* holds what matched. Or, if the parameter is of a non-object type, one of *the <type> understood* variables is used, such as *the topic understood*, *the time understood*, *the number understood*, etc.

For actions with two parameters, a preposition follows. While the player doesn't always need to enter a preposition – consider GIVE BOB THE TOY – the programmer does. Since the action is named *inserting it into*, our preposition is *into*. (The *it* in the action name is a placeholder.)

The second noun follows, but we could have stuck the colon here to end the preamble. Instead, *the jeans,* or any variable or set-description. *Something* is synonymous with *a thing*, and is the usual catch-all. The variable *the second noun*, or *the <type> understood*, holds what matched.

But wait, there's more. The word *when* means *if*, and any valid condition can follow it. *The time of day* is a time variable in the library, automatically advanced one minute per understood player command. The “is before” means “is less than”.

Finally, the colon marks the end of the rule preamble – the regex-like part of the rule – and the

start of the imperative code block. (One piece of syntactic sugar: a comma can be substituted for the colon if the imperative code block is a single line, and the rulebook begins with *before, after, instead of, when*, or *every turn*. Very pretty code can result.) Here our block consists of a single print statement. The trailing "instead" is another bit of syntactic sugar for appending the statement *rule fails*. Since the rule failed, the rulebook also fails, and execution skips ahead to the *every turn* phase of the turn. Since that skips the *carry out* rules, the action does nothing except print our denial message in the situation our preamble describes.

All rules in the language follow that pattern. As for rulebooks, they are simpler. Every rule ends in one of the statements *rule succeeds*, *rule fails*, or *make no decision*. If we don't specifically use one of those three statements at the end of the rule, the rulebook's default is automatically appended. *Rule fails* and *rule succeeds* both end the rulebook as well – whatever task was given to the rulebook has completed. *Make no decision* tells the rulebook to try the next applicable rule, and is the default for any rulebook that runs all its rules. *Make no decision* is the default for *when play begins, before*, *check*, *carry out*, *report*, *every turn*, and almost every other standard rulebook. Of course, *instead* defaults to *rule fails.* *After* and each activity's *for* default to *rule succeeds*.

So, some examples with actions. Square brackets outside a text string denote comments.

```
Report going when the current manager is in the location: say "See ya later, [the
current manager]." instead.

Carry out examining the cell phone for the first time: [ once, twice, and for the Nth
time can end a preamble ]
    now the player worries about Jesse;   [ worries about would be a relation ]
    now the cell phone is fingerprinty.    [ fingerprinty a boolean property ]
```

Some examples with other, background events.

```
When play begins, say "Once upon a time..."

After reading a command when the player's command includes
"quickly/quietly/slowly", say "I cannot understand adverbs in commands." instead.
[ slashes mean or ]

Before printing the name of Bob, say "Mr. "
```

Within a rulebook, rules are sorted according to the specificness of the preamble. So, *instead of examining someone suspicious* will always be tried before *instead of examining someone* no matter their source code order. All the various restrictions must still be satisfied of course. This just determines what rules are tried first, and if multiple rules share the same specificness, they are considered in source code order. Although this setup sounds frighteningly like building atop shifting sands, empirically it has worked better than anticipated. This is likely because rules belong to rulebooks, and only one rulebook is operating at any one time, in much the same way that only one function operates at one time: the caller is "on hold". And unlike Prolog's sea of rules, one rulebook is small enough to remember at once. But when in doubt, there is the Index.

Found in the IDE, the Index is the automatically-generated author's documentation. The Index button is labeled vertically along the right edges of both panes, and will be empty until the first successful compilation. The various tabs of the Index list all the constructs in the author's work as of the latest compile. It's possible to learn a lot just by reading it. The Actions tab lists all the actions the game knows, both built-in and author-written, and lists the player words that trigger each. Clicking the magnifying glass icon next to an action name will show a detail page listing all the rules that can apply to it. The Contents tab lists all variables and tables. The Kinds tab not only contains the object hierarchy including all instantiations, but also all other types and under what generic types they fall. The Phrasebook tab holds procedures, functions, and relations. The Rules tab holds only rules that do not apply to actions or scenes. The Scenes tab holds detail on scenes and scheduled events, while the World tab holds the room map.

Now that we've covered rules and the index that would be new to many programmers, we can speed up our tour. The only thing left to start creating games is how to give the player synonyms for objects and actions. Set-descriptions can be within the square brackets here, but generally it's better to allow much at this parsing stage, and use *check* rules to print better denial messages for attempts to wear doorknobs or eat Ferraris.

> **Understand** “squint at [something]” **as** examining.
> **Understand** “dash [direction]” **as** going.
> **Understand** “stick [something carried by the player] down/into [something]” **as** inserting it into.
> **Understand** “Robert” or “Rob” **as** Bob.
> **Understand** “dude” **as** a man. [narrows down the referent to objects of class Man]

Conditions can be attached. *The item described* is the *self* or *this* variable used in other languages. For properties, a shortcut obeys the current state of the property.

> **Understand** “bastard” or “rat bastard” **as** Bob **when** the time of day is before the deadline.
> **Understand** “fingerprinty” **as** a thing **when** the item described is fingerprinty.
> **Understand** the fingerprinty **property as describing** a thing. [better for properties]
> **Understand** “smudged” **as** fingerprinty.

Functions are called to-decide phrases. Because their return values cannot be thrown away, they will always appear within a larger statement, and so, are named after noun phrases. Below, **bold type** shows keywords in the definition before the colon. The colon is where the function body starts. *Decide on* is the return statement. The return value comes just before *is*. Slashes denote a choice of word (pick exactly one), or when combined with the – double-dash, the word is optional. Python indentation, or *begin* and *end if/repeat/while*, denote blocks, but begin-end and Python indentation cannot be mixed with each other in the same function.

> **To decide which** number **is** (x – a number) smooshed with (y – number): decide on x multiplied by y.

To decide what object **is** my toy:
    if the Ferarri is in the location, decide on the Ferarri;
    otherwise decide on nothing.

To decide which person **is** who wears the pants around/in the/-- house of (p – an unhappy person):
    if p is: [Python style is required to use the switch statement]
        – Bob: decide on Edith;
        – Jose: decide on Maria;
        – otherwise: decide on the mother of p.

Boolean (*truth state)* functions use *to decide if* and are named after clauses. *Yes* and *no* are the phrase versions of values *true* and *false*, while *whether or not* typecasts between.

To decide if (p – a person) will be done by (deadline – a time):
    if p is Bob, yes;
    otherwise decide on whether or not the time of day plus five hours is before the deadline.

Adjectives can be used as part of set-descriptions, which the above cannot. The pronoun *it* refers to the object in question here, but an explicit *called* parenthetical can name it. One-liners can use *if* rather than the colon. The antonym is defined by **rather than** but is optional.

Definition: a person **is** overloaded **rather than** good **if** the number of workorders expected of it is at least five.

Definition: a person **(called** the workerbee**) is** efficient **rather than** slow:
    repeat with item **running through** every workorder expected of the workerbee:
        if the completion time of the item is greater than one hour, decide no;
    decide yes.

On the downside, parameters cannot sit right next to one another. A word must come between. But, the type of a parameter isn't just a type or class, but anything from a specific value to a full-on set-description. As with rules, Inform chooses the most specific applicable phrase, and in the case of a phrase calling itself, prefers not to recurse if possible. (This is mainly a safety feature for the lay target audience.) So for the following procedures – functions which return no value, and sometimes called to-phrases in Inform because *to* is the only keyword they have in common – if Jenny doesn't love the player, then invoking her name causes execution to ping-pong between both applicable phrases.

To tell (lover – a person) I love him/her/them recursively:
    say “I love you, [lover].”;
    tell the lover I love them recursively.

To tell (lover – a person which does not love the player) I love him/her/them recursively:
    say “I secretly love you, [lover].”;
    tell the lover I love them recursively.

To tell (lover – Bob) I love him/her/them recursively: say "Er, how's [abrupt subject change] coming along?"

Say-phrases are called from within text strings. *If* constructions are standard, but cannot nest: "[if Jenny loves Bob]What? Why?[otherwise if Bob loves Jenny]Welcome to the club. [otherwise]Won't she give me her number?[end if]"

To say abrupt subject change:
    if the current weather conditions are not mostly sunny:
        say "the weather";
    otherwise if Edith is not in the location:
        say "you and Edith";
    otherwise:
        say "the Chicago Cubs".

To say looks like it's (w – a weather condition):
    if w is nasty, say "blowing up a storm"; otherwise say w.

A blank line, or a period substituted for a semicolon, ends a phrase.

Named values (known as enums in C) can do most anything objects can do, and needn't be all defined at once. *A weather condition is a kind of value. Some weather conditions are rainy, partly cloudy, mostly sunny, about to snow, and nasty. A weather condition is a tornado warning. A metal is a kind of value. The metals are copper, tin, iron, aluminum, platinum, nickel, gold, and silver.* One standard kind of value receives special attention: scenes. With a *during* preamble phrase and its own SCENES testing command, scenes are a powerful organizing principal in an interactive fiction.

The denouncement is a scene. The denouncement begins when the climax ends. Instead of taking something during the denouncement, say "But the life of a kleptomaniac seems so empty now."

Boolean properties. *A person can be happy or unhappy. A person can be resourceful. A person is usually not resourceful. A weather condition can be later in the evening. A weather condition is usually not later in the evening.*

Valued properties. *A person has a person called its mother. A weather condition has a time called the predicted duration. The predicted duration of a weather condition is usually five hours. Bob's mother is Agnes.*

Subclassing. *A boy is a kind of man. A girl is a kind of woman.* Statically-allocated means no constructors or destructors. Inform can insert new classes between existing classes. *A conveyance is a kind of container. A vehicle is a kind of conveyance.* This does not contradict the standard *a vehicle is a kind of container.*

Timed events are rules that self-invoke at a given time or in a given number of turns. They fire only once, but named ones can re-schedule themselves.

At 6:30 am: **say** “Your alarm clock screams you awake.”

At the time when an email arrives:
    say “You've got mail.”; an email arrives **in** seven **turns from now**.

At the time when the alarm clock screams:
    say “Your alarm clock screams you awake.”;
    the alarm clock screams **at** 6:30 am.

Activities are multi-rulebook events not generated by the player. They exist merely so the library (or authored extensions) may provide hooks for important events. The three phases of an activity are *before, for,* and *after. Printing the name, supplying a missing noun, printing a parser error,* and *reading a command* are the most useful. Other standard rulebooks which are not full-on activities are *does the player mean*, *persuasion*, and *unsuccessful attempt by*.

After printing the name of Bob: say “ Jr.”
After reading a command when the player's command includes “please/thanks”, cut the matched text.
After reading a command when the player's command matches “take nap”, replace the matched text with “sleep”.
For printing a parser error when the latest parser error is I beg your pardon: say “Please type something in.”

Does the player mean taking the train: it is very unlikely. [getting on or entering the train, perhaps]

Arrays currently have little support since who would want to read about a hundred perfectly identical things? Still, it's possible to instantiate numerous unnamed objects, but only if we specifically create a subclass for it. *A coin is a kind of thing. Seven coins are in the sofa. A light is a kind of thing. There are four lights.* While the player can TAKE THREE COINS, the programmer resorts to set-descriptions and *a random*, such as *a random off-stage coin.* (*A random* is a decide phrase, not an adjective. The *a* is required. *Off-stage* is when *location* is *nothing*.) Tables are 2D arrays, mostly, and can instantiate a slew of objects or named values at once. Tabs are required between columns. A – double-dash is a blank entry.

A workorder is a kind of value. Some workorders are defined by the table of tasks.

Table of tasks
workorder	employee
send invoices	Bob
check addresses	–
restock	Bob

And we close off the firehose with relations. Object-oriented programming is built on two object-to-object relations, is-a and has-a. Single inheritance is one-to-various, while multiple-inheritance is various-to-various. Object properties in general are each a various-to-one relation. Indeed, that's frequently how Inform implements them: the *called* parenthetical allows the *noun of noun* syntax of properties as an alternative to the *noun verb noun* syntax of relations. But *called* is restricted to the singular, so the *reversed* word may be needed.

Finally, Inform already knows the singular (-s), plural (root), past (-ed), past participle (-en), and present participle (-ing) forms of *be*, so the large parenthetical is omitted for such.

Single inheritance **relates** one thing **(called** the parent**) to** various things.
The verb to be the superclass of **implies** the single inheritance relation.
The verb to inherit from **(he** inherits from, **they** inherit from, **he** inherited from, **it** is inherited, **he** is inheriting from**) implies** the **reversed** single inheritance relation.

now dog **inherits from** animal; [These are synonymous, but note we're not using Inform's actual class hierarchy.]
now the **parent** of dog **is** animal; [The real class hierarchy is immutable at runtime.]
now animal **is the superclass of** dog;

Assignment **relates** various workorders **to** one person **(called** the employee**)**.
The verb to be expected of **implies** the assignment relation.

Class And Prejudice

Here's the built-in class hierarchy, a total of sixteen classes. (Also, where other languages say "class", Inform says "kind".) Inform purposely keeps its library lean.

object
- direction
- room
- region
- thing
  - door
  - container
    - vehicle
    - player's holdall
  - supporter
  - backdrop
  - device
  - person
    - man
    - woman
    - animal

A quick rundown: Room is a discrete location, a place; Region is a container for Rooms. The class Container means an in-game prop, such as a backpack or hamster cage. A Supporter is a chair, table, mantel, or other horizontal surface Bob can place things on top of. The player's holdall is usually a singleton, as it's a container without load limits. A instance of Door can explicitly connect Rooms, and can be open & closed if not also locked & unlocked. Direction does NOT connect rooms -- relations do that -- but is used in code to reference the same. Backdrop always has the boolean property *scenery* set, and is used for things like the sun in the sky, the faint but ever-present sound of a nearby creek, and other non-portable objects that need to remain in the parser's scope while the protagonist travels across several Rooms. Device is something that can be switched on/off. Animal is treated as a kind of person, just as pets are.

The positioning of Animal is our first hint that we're stepping out of a scientific worldview, and into a humanistic one. Likewise, the purpose of the language as a whole is to produce works of art, not software tools.

Making a subclass is straightforward. Note that *are* can replace *is*, and most any article can replace *a*.

> An archway **is a kind of** *door*.
> An archway **has** *a number* **called** the horizontal clearance. It **is usually** 6.
> An archway **is always** *open*.
> An archway **can be** magic **or** mundane. An archway **is usually** not magic.

This subclasses archway from door. It gives it a new numeric property call the *horizontal*

clearance with an initial value that can be overridden by a particular instance. Then it permanently sets the pre-existing open/closed property to *open* so attempts to later code a closed archway will result in a compiler error. Finally, it gives it a new anonymous boolean property with named values *magic* and *mundane* (as opposed to a property called "magic" with values "true" and "false"). It initializes this to *mundane*. We could have just said *an archway can be magic* and it would still work, but naming the antonym usually improves readability.

One instance of *person* is always provided for us: *yourself*. *The player* is a person variable initialized to *yourself* unless defined otherwise. (Only the player refers to the avatar as ME.) One instance of Object (the root-level class) is always provided: *nothing,* the nil pointer. Occasionally synonyms nowhere, nobody, no-one, and no one may be used, but it's usually a special-case syntax.

```
The player is Bob.                  [ the yourself object still exists, though unused ]
[...]; now the player is Bob;             [ imperatively, at runtime ]
```

Object names can unfortunately occlude one another. If we have an object "car", and then an object "car key", it frequently happens that the parser cannot refer to the car. Inform supplies objects with a property, *privately-named*, which prevents automatically exporting an object's source code name to the player's parser. Instead, an explicit *understand* line will be required.

The articles *a, an,* and *the*, and including *some* when used similarly, are only mostly ignored in Inform. Inform will mimic in the game's output the articles used in the source, with the object's first mention in the source given extra weight in case of inconsistent usage. Objects created with *some* will be treated as mass nouns, and objects whose names are always capitalized in the source are capitalized as proper names.

Inform does not (currently) support arrays, but it does have a couple of alternatives. A later section shows us tables, which are similar to 2D arrays. But here, we can instantiate up to a hundred indistinguishable instances of a kind at once. A subclass is required for this, but that is easy enough to arrange.

```
A coin is a kind of thing. 55 coins are in the couch. Three coins are carried by the
player. There are seven [more] coins.
```

That last definition creates objects but leaves them *off-stage*. The commented-out word “more” is only placed there for clarification. It is especially useful in cases like: *There are seven coins. Three coins are in the couch*. Those sentences might imply there are a total of seven coins in the world, rather than ten, seven being *off-stage* and three *on-stage*.

There's a few things to know about unnamed instances. We must subclass before instantiating or else we'll have a single object called "55 coins". We can't reference “the thirty-second coin” like we could with an array, but instead must use set-descriptions with *a random* to grab one: *try taking a random tarnished coin which is not carried by the player*. And it's up to us to verify such an instance exists. We can't specify a quantity of them as in *now the player carries three coins* or *try taking three coins* because Inform won't know which three coins, but we can manipulate all of them as in *now the player carries every coin*. So, instantiate away, but as far as Inform 7 is concerned, there's only zero, one, or infinite coins.

Our player can TAKE THREE COINS if the action's *understand* line applies to *[things],* plural, as *taking* and *dropping* already do. Furthermore, we can expose the various properties (adjectives) of our new subclass to the player's parser, giving the player the ability to differentiate with TAKE THREE TARNISHED COPPERS. *As referring to* exposes it as a pure adjective, while *as describing* exposes it as both adjective and noun.

```
A coin can be pristine or tarnished.
A metal is a kind of value. The metals are copper, silver, and gold. A coin has a metal.
Understand "coppers" as copper. Understand "silvers" as silver. Understand "golds" as
gold.

Understand the tarnished property as referring to a coin.    [ tarnished or pristine by
itself won't grab a coin ]
Understand the metal property as describing a coin.          [ but copper, silver, or
gold will ]
```

Though we wouldn't normally expect the player's parser to be more expressive than the programmer's, in the case of taking three coins, the runtime always has the option of interactively asking for details ("Which did you mean?") while we do not.

The Coding Imperative

Here are examples of all the basic imperatives. Read the print statements within.

```
let Z be 5;                     [ Local variable declaration; Type is usually inferred. ]
let t be indexed text;          [ Explicitly stating type, for next line. ]
let t be “Hello world”;                 [ This will be indexed text instead of text. ]
now X is 5;                     [ assignment ]
now every V W is X Y Z;         [ assign properties X, Y, and Z to all instances of W
        which already possess V. ]

if X > Y, say "A comma is synonymous for 'then'.";
otherwise say "'Otherwise' (or 'else') must be a one-liner if the 'if' was a one-liner. No
        punctuation follows the word.";

if X > Y begin;
        say "'Begin' requires a semicolon of its own.";
otherwise;
        say "As does the matching 'otherwise' and 'end'.";
end if;

if X > Y:
        say "Python-esque style is OK as long as we don't mix the two styles in the
                same function.";
otherwise:
        say "Posting code to internet forums usually corrupts the tabs, but tables
                require tabs anyway.";

if X is greater than Y begin;
        say "This is a else-if chain. Also, notice that relational operators can be spelled
                out.";
otherwise if X is Y;
        say "Only a semicolon is found for the trailing conditionals.";
otherwise;
        say "The final otherwise is the same as usual.";
end if;

if X is greater than Y:
        say "This is a else-if chain in Python style.";
otherwise if X = Y:
        say "The punctuation here is more regular: always a colon, no matter what.";
otherwise:
        say "The = is rarely seen by itself, in practice; 'is' is easier to type.";

unless X <= Y, say "'Unless' means 'if not'. Also, inequalities are written >= or <=,
        never => or =<.";
```

```
if X is:
	-- 1: say "A switch statement masquerades as an if statement. 'Unless' cannot be
		used.";
	-- 2: say "Python style only.";
	-- otherwise: say "There is no fall-through between cases, and no goto to
		restore it.";

if my favorite toy is:
	-- the red Ferrari:
		say "Numbers and objects both can be used in switches.";
	-- the jeans:
		say "Also note the further indentation of the cases, and their subsequent
			lines.";


repeat with X running from 1 to 10:
	say "'Repeat' is the usual loop construct. It has [X] forms. This enumerated one
		isn't used much.";

repeat with clothing running through every wearable thing:
	say "The 'description' type describes a set of objects.";
	if the clothing is the pair of shoes, say "Also, 'every', 'each', or 'all' aren't
		required here but may read better.";

repeat with target running through all limbs:
	if the target is the head, say "We can repeat through named values, scenes,
		action names, etc. just as easily.";

repeat with programmer running through the men who are in a lighted room (called the
	mainframe's area):
	say "[The programmer] in [the mainframe's area] says we can do some pretty
		complex stuff with 'descriptions'.";
	next; [ 'next' starts the loop over at the next iteration. C calls this 'continue' ]
	break; [ 'break' kills the loop immediately ]

repeat through the table of designs:
	say "Tables are 2D arrays.  We'll look at them in detail in their own section.";

while X is at least Y begin;
	say "While loops are rarely seen in practice.";
end while;
```

Semicolons divide statements as usual, but the last semicolon should instead be a period. Alternately, the last statement should be followed by a blank line. This is how the end of a function is signified.

Inform calls the above imperatives, and other things like them, *phrases*. Functions are also called phrases, because they are invoked similarly. But rules are different; they invoke themselves based on game events and other worldsim situations. Here's some quick examples

just so we'll have somewhere to try out our imperative code.

```
When play begins: say "Hello world!".
Every turn: say "La-dee-da."
Instead of taking yourself, say "You pull yourself up by your bootstraps and read on."
```

Remember to instantiate a room.

Boolean Adjectives

Adjectives, and the set-description type that invokes them, are Inform's big win over the COBOL and HyperTalk families of natural language programming languages. Typically used for objects, a description uses combinations of adjectives and/or relations with a class to define a set of instantiations. This is responsible for a great deal of Inform's conciseness over traditional programming languages. First we'll look at how to define the adjectives.

There are a few ways to define a boolean adjective. The first creates an anonymous boolean property with named values.

> A thing **can be** spiffy.
> A person **can be** grumpy **or** happy.
>
> **now** the iPod **is** not spiffy;
> **if** Mary **is** happy, **now** Bob **is** grumpy;

Inform of course understands *not spiffy* but it also understands *not grumpy* as synonymous with *happy* and vice-versa. Also, since *person* is subclassed from *thing*, a person can also be spiffy. Frequently in OOPLs, new properties and methods cannot be added to a pre-existing class because doing so would break pre-existing code that uses it. Subclassing is required to add such embellishments. But since Inform isn't designed for reusable code or team coding, it allows directly modifying classes, and those changes propagate down the hierarchy regardless whether they are built-in or not. The allowed modifications are additive only, however. (Though sectional replacement of source code alleviates even that.)

The second way to define an adjective works like adding a boolean-returning method to a class. It also has a one-liner version using *if* instead of a second colon:

> **Definition:** a person **is** kinda dull**:** [...]; decide no.
> **Definition:** a person **is** unlikable **if** it is kinda dull or it is grumpy.
> **Definition:** a person **(called** the academic**) is** laconic **rather than** chatty **if** the academic is [...].

The pronoun *it* refers to the object on which the method is called, where traditional languages use "self" or "this" or some such. This can be customized with the *called* parenthetical. An antonym may be defined with the *rather than* phrase. *Definition:* adjectives aren't restricted to objects. Named values are also common here.

(The third way, using a plain *to decide if* boolean function, loses the flexibility of appearing in set-descriptions. But, being a function, it can take any number of arguments, and may use the powerful parameter-matching logic of same. We will see them soon.)

The adjectives are checked the same way regardless whether they are data or calculated, so changing an adjective from a variable implementation to a calculated implementation (or vice-versa) only requires adding or removing any lines that explicitly set the adjective's value. Client code that merely checks the value needn't change.

```
repeat with associate running through every chatty not grumpy spiffy person:
        say "Hi [associate]."
```

Patterned Procedures

A procedure's "name" in Inform 7 is not an exact match of a single identifier, but a simple textual pattern. The word *to* starts the definition, and parenthesis enclose the local variable name, a hyphen, and the type. A colon ends the "name" and the imperative code block begins.

```
To plainly greet (friend - a person):
    say "Hi [friend]."
```

The above is invoked readably.

```
plainly greet Dr. Muller;
```

The / forward slash can be used to allow synonyms for a word by acting as a high-precedence disjunction: choose exactly one. Its effect ends at the first space. The -- double-dash in a disjunction means the word is optional: choose at most one.

```
To ponder/mull over/-- (good point - a thing) for (awhile - a time) as (ponderer - a person):
    say "[Ponderer] sits back for about [awhile]. 'Hm, [good point] is a very good point.'"
```

The following invocations are all equivalent.

```
ponder the best idea yet for 7 minutes as Dr. Muller;
ponder over the best idea yet for 7 minutes as Dr. Muller;
mull best idea yet for 7 minutes as Muller;
mull over the best idea yet for 7 minutes as Dr. Muller;
```

A parameter can immediately follow the *to* that begins the definition, but parameters cannot sit immediately side-by-side.

```
To (ponderer - a person) ponders (good point - a thing):      [ok]
To ponder for (awhile - a time) (good point - a thing):    [error!]
```

The types of the parameters needn't be a single type, or even a type at all. Set-descriptions and particular values both may narrow the scope of the input. Like actual rules, this is a handy feature for special cases.

```
To plainly greet (foo - people which are friends with the player):
To plainly greet (foo - a resourceful person):
To plainly greet (foo - nothing): say "Run-time error: called 'plainly greet' with nothing."
```

While set-descriptions narrow the accepted range of input, generic types allow one phrase to cover many different types. The type of a parameter itself can be captured and used elsewhere in the definition and/or code block via the phrase *value of kind K* and similar

variables *L, M,* etc. *Sayable value*, *word value*, *pointer value*, *arithmetic value, enumerated value*, and the most generic of them all, *value*, are the known generic types. They can be used with or without *of kind K*.

To debug-print the differences between (construct one – a sayable value of kind K) and (construct two – K):

Let creates a local variable. *Let* is also how we create flexibly-sized types like *list* and *indexed text*, since they cannot be statically allocated and the language does not allow dynamic allocation.

```
    let X be 5;
    let the typical exclamation be "That's cool!";
    let the articles of clothing be the list of things worn by Bob;
    let M be { the red Ferrari, the pair of shoes };
    let the modifiable exclamation be indexed text;      [  These work...   ]
    let the modifiable exclamation be "That's cool!";    [      ...as a pair.   ]
```

Stop is the plain-jane return statement. It isn't used much, partly because of some other synonyms for return, and partly as fallout from the rules-based structure of the language.

A final natural-language feature cloaks a bitfield as a series of comma-separated sub-phrases.

```
    To go hiking, into the woods or up the mountain:
        if into the woods then say "Watch out for badgers.";
        if up the mountain then say "Better take your compass.";
        say "You go hiking."
```

Clever naming not only affords client code that is easy to read, but also creates library invocations that are easy to make a half-remembered guess at. The latter is, in practice, a wonderful time-saver. When we define functions, we should take client code readability into account. For example, we needn't add the articles in front of a parameter, because the parameter itself will eat it.

```
    To ponder the/an/a/-- (nefarious plans - a rulebook):        [ unnecessary ]
    To ponder (nefarious plans - a rulebook):                        [ better ]
```

But we should explicitly add the articles if it occurs elsewhere, such as this example that pretends to understand an adjective in some cases.

```
    To ponder the/an/a/-- foiled/new/-- (nefarious plans - a rulebook):        [  necessary ]
```

To phrases tend not to be used too much, for the similar reason that methods in a OO language tend to reduce global function use. Instead, Inform 7 has rules, grouped into rulebooks, which we'll get to shortly. To-phrases do have a nicer invocation syntax and essentially unlimited arity to recommend them over rules, but they lack some of the flexibility of rulebooks, as we will soon see.

Functions Decide on a Value

Because strong-typing requires returned values to be used, functions will always be used in a larger statement. Hence their names tend to be noun phrases rather than sentences.

```
To decide which room is my favorite place:  [...]; decide on My Bedroom.
To decide what person is the brother to/of (sibling - a person):  [...]; decide sibling.
To decide which object is my fabulous doodad:  decide on a random thing.

ponder the best idea in my favorite place;
if the brother to the noun is not the noun, say "[The noun] has a brother, [the
brother of the noun].";
if my fabulous doodad is nothing, say "I'm fresh out of fabulous.";
```

To decide which, or synonymously, *to decide what*, begin the definition. The return value follows, and then *is*. The function's name is only that part between *is* and the colon. The return statement is *decide on*. Due to strong-typing and *nothing* being pronounced an instantiation of class *object*, we cannot *decide on nothing* except when the function's return value is type object.

Boolean functions must use the slightly different *whether/if* variation, and the name lies between the *whether* or *if* and the colon. Since they are invoked from if statements and rule headers, their names are usually clauses sans subordinating conjunction.

```
To decide whether (pants - a thing) is/are on fire:
        decide on whether or not a random chance of 1 in 2 succeeds.

if the brother of the noun is on fire, say "That's gonna leave a mark."
```

The phrase *whether or not* typecasts an if-condition (such as if "a random chance of M in N succeeds") to a truth state (boolean), which can then be returned.

Say Phrases

It is so common to slightly vary some prose for a given situation that Inform specifically provides for procedures called from within a *say* statement's prose. Say-phrases are in a box separate from to-phrase procedures and to-decide functions, but otherwise work identically. They are invoked by square brackets within a text string.

Gendered pronouns are a common case, and most are built-in.

```
To say He-She for (P - a person):
    if P is plural:
        say "They";
    otherwise:
        if P is female, say "She";
        otherwise say "He";

To say (P - a person) mulls/ponders --/over (idea - a thing):

[...]; say "[He-She for Chris] glances at you[Chris ponders tar-and-feathering].";
```

Putting a bare object or variable name within the square brackets prints the entity's name or variable's value, respectively. This works for nearly every type in the language, though can usually be overridden like so.

```
To say (code - a rule): abide by the code.
```

That would execute the rule or rulebook from within prose, rather than printing something.

```
say "Chris seems to make a decision.[the formulate plans rules] But you don't know what.";
```

Inform ships with a number of basic imperatives for say phrases. The docs have the full list, and the Extensions chapter has information on creating new multi-part To Say constructions.

```
say "He put on [if the jeans are stained]yesterday's[otherwise]his[end if] jeans.";

say "The weather was [one of]rainy[or]sunny[or]windy[at random].";
```

Types of Types

We know defining a class that implements a list of numbers is tedious, but not as tedious as doing it again for strings, objects, and any other type the language supports. Most languages nowadays provide for this problem, such as the templates of C++. Inform too offers generic types. *Arithmetic value* allows numbers and units. *Pointer value* includes indexed text, stored actions, and other variable-sized chunks of memory. *Word value* covers all non-pointer values, including objects. *Sayable value* is any type allowed in a print statement, which is almost anything. *Enumerated value* is named values, which includes scenes. And finally, *value* is all of them: just about anything that can be passed as a parameter. The Kinds Index in the IDE shows what types belong under which umbrellas. We can use them to define a phrase like this.

To place angle brackets around (foobar - **a sayable value**): say "<<[foobar]>>".

To print the elements of (stuff - ***a list of*** **sayable values**): say "The list contains [stuff]."

Kind variables tie two or more parameters to the same type. This is particularly useful when one parameter is an aggregate type, and the other is the type being put into, brought out of, or compared to an element of, that aggregate. Kind variables are always a single capitalized letter, and traditionally use K and L.

To decide which **K** is the initial contents of (stuff - ***a list of*** **arithmetic values of kind K**): ...

To we will ask if (col - a **K** valued table column) is (data - **a word value of kind K**): ....

The words *value of kind* must precede exactly one of the Ks. That parameter will be the one that declares what K holds, so the other parameter's input will be expected to match. For example, the following line means the second parameter decides what K is, so the compiler will search for a table column of that type.

To we will ask if (col - a **K** valued table column) is (data – **a value of kind K**): ....

While this means the first parameter's type sets K, and the second will be interpreted as the type in question.

To we will ask if (col - **a value of kind K** valued table column) is (data - **K**): ....

It makes a difference. A very few types are implicitly casted between each other, such as text and indexed text, and whether K is text or indexed text will likely matter in the body of the function. Other times, the compiler may have a choice of constructs with the same name but different types. Or, the same words of source text may have radically different interpretations depending on what type the compiler expects it to yield. And one final note: the return value cannot set the kind variable. It may use one, as the above example with initial contents, but

cannot declare it.

The parameter *name of kind of value* is an interesting case because, rather than accepting a particular instantiation of a restricting type, it asks for the source code's name of a type, such as *object* or *room* or *weather condition*. K is set to the type, and then K is used in the body of the function in the same places and same ways as the word itself would have appeared. The feature is also very useful in typecasting, via Inform 6.

```
To rattle off all the (name of kind of value K):
        repeat with x running through K:
                say "[x], "
```

To decide which K is the (mystery - a value) as a (**name of kind of value** K): (-{mystery} -).

We can then write:

```
rattle off all the scenes;
rattle off all the tattoos;
rattle off all the rooms;
let whatsit be foobar as a tattoo;
let whosit be foobar as a person;
let howsit  be foobar as a rule;
let whensit be foobar as a scene;
```

And so on. Most OO languages tend to treat everything as an instantiation of a class. In Inform, many constructs are not objects, but the type system allows us to use the same function on everything just the same. (Note that parameter types *condition* and *action* are still a special case. They can only be used in a phrase when the body is written in Inform 6. The later chapter on Inform 6 covers them.)

To-phrases and relations can be passed as parameters, but still strongly-typed. Examples of the type of a phrase would be:

```
a phrase (length, length) -> nothing
a phrase nothing -> number
a phrase (number, number) -> number
```

And for relations:

```
a one-to-various relation of people to cars          [ asymmetric is always assumed ]
a symmetric one-to-one relation of people            ["to people" is assumed]
an equivalence relation of people
```

One thing Inform cannot do is an indefinite number of parameters, such as the printf function in the C programming language. And because Inform is strongly typed, there must be a slew of phrases that apply a passed-in phrase to 1, 2, or 3 inputs, and returning a value or not. For phrases that return a value, use one of *(phrase) applied*, or *(phrase) applied to (value)*, or *(phrase) applied to (value1) and (value2)*, up to three parameters, just like rules. For phrases

that do not return a value, *apply (phrase)*, or *apply (phrase) to (value1)*, etc.

In lieu of indefinite parameter counts, Inform can pass around lists of whatever, and has the higher-order functions. Map, as *(phrase) applied to (list)*, takes a list, runs the passed-in phrase on each element in turn, and returns the new list. Filter, as *filter to (description) of (list)*, returns a smaller list than the one passed in, having removed what elements don't fit the set-description. And reduce, as *(phrase) reduction of (list)*, which returns a single value. However, these only emulate the syntax of functional programming. Without lazy evaluation or anonymous functions, this style of programming leaves much to be desired, and cannot at all deal with lists of infinite size in the common generator-consumer pattern.

Sweet Relations

In lieu of numerical relationships, qualitative either/or relationships frequent interactive fiction. Relations disguise 2D boolean arrays behind some of the best syntactic sugar in the language – namely, behind verbs. Relations are written with the infix form of English verbs. Asymmetric (*X to Y*) and symmetric (*X to [one] another* or *X to each other*) relations are possible, with one-to-one, one-to-various, various-to-one, and various-to-various flavors, plus the equivalence relation (*[various] X to each other in groups*). The *called* parenthetical, only on a singular side, allows property syntax as well.

```
Marriage relates one person to another (called the spouse).
[ one-to-one symmetric ]
The verb to be married to implies the marriage relation.

Bob is married to Jane.
When play begins, if Jane is married to someone, say "Jane's husband is [the spouse of
Jane]."        [  prints Bob  ]
```

Symmetric relations are always of type X to type X. So marriage cannot be "one man to one woman" nor can friends be anything but person to person.

```
Friendship relates people to each other.
[various-to-various, symmetric]
The verb to be friends with implies the friendship relation.

if the brother of the noun is married to the second noun, now the noun is friends with
the second noun;
repeat with pal running through every person who is friends with the brother of the
noun:
```

When we define a verb like *to be X*, Inform can automatically conjugate it. But for *to X* we supply the other five forms in parenthesis. They are, in order, the singular (-s), plural (root), past (-ed), past participle (-en), and present participle (-ing). At least either the singular or plural must be supplied. *He/she/it* may be used interchangeably, but *they* has no synonym.

```
Like-minded relates various people to each other in groups.
[ an equivalence relation ]
The verb to draw (he draws, they draw, it drew, he is drawn, she is drawing) implies
the like-minded relation.
```

Asymmetric relations may find a *reversed* synonym useful in set-descriptions.

```
Teenage love relates various people to various people.
[ tragically, an asymmetric relation ]
The verb to be in love with implies the teenage love relation.
The verb to be lusted after implies the reversed teenage love relation.
```

Additionally, we can define the verb as *to be able to X*, which means the relation is used like *can X*. The parenthetical has only the past participle (-en) form, and is required.

> Memorability **relates** *various* people **to** *various* things.
> **The verb to** *be able to* remember (he is remembered) **implies** the memorability relation.
>
> Bob is a man. The time he spent in jail is a thing.
> Bob **can remember** the time he spent in jail.

The *to be able to...* relations (only) can be stated in the passive voice as well. So *Bob can remember jailtime* may be stated *jailtime can be remembered by Bob*. Effectively, the *reversed* version is provided automatically. The phrasebook part of the Index shows the available wordings.

Though only an asymmetric various-to-various relation actually needs a whole 2D array, the other kinds of relations are some sort of subset of one. Any relation with a singular is frequently implemented as a property, as are equivalence relations. Relations on very large (text) or infinite (numbers) domains will use sparse array implementations, such as properties whose type is a list of some sort.

Much like boolean adjectives, relations can be implemented as calculation rather than storage.

> Siblinghood **relates** a person **(called** X**) to** a person **(called** Y**) when** X is the brother of Y or X is the sister of Y.
> **The verb to** be a sibling of **implies** the siblinghood relation.

The syntax to check whether a relation holds between two things is the same regardless the implementation, so swapping between them only requires fixing up wherever the relation was explicitly set between things. It is worth nothing that the relativistic relations of math are implemented this way ("The verb to be greater than implies the...") as well as almost all spatial relations (under, on, in, northeast of, etc.)

Relations can be put into lists, or passed into functions.

> To decide whether (relationship – a **symmetric various-to-various relation of people to people**) is a good reason for trusting (P – a person): decide on whether or not ***relationship* relates *P* to *the player***.

Asymmetric is assumed, so only *symmetric* is defined. The *one-to-various* and flavors obviously cannot be used with *equivalence*. Finally, *empty* describes the default condition. The phrase *relates...to...* can be used in *if* and *now* statements. In an *if*, types and negation can be used: *if Bob **does not relate to** someone **by** the love relation.*

Note that the above phrases and the following phrases are indeed *phrases*, not a set-description (which the following resemble) nor a three-way relation (which the former resemble, assuming Inform could even do multi-way relations).

Various functions provide lookup. These are especially handy when dealing with domains as

large as text or numbers. Unfortunately, the compiler cannot check them at compile-time. Runtime type errors result if arguments are swapped.

a Y which X relates to by R [for various-to-one or one-to-one]
a X which relates to Y by R [for one-to-various or one-to-one]

list of Ys which X relates to by R [for various and equivalence]
list of Xs which relate to Y by R [for various and equivalence]
[Warning: the above two are new and very buggy.]

list of Xs which R relates [for finding everything on the left hand side of relation R]
list of Ys which R relates to [for finding everything on the right hand side of relation R]

Pathfinding uses relations.

let X be **the number of steps via** the acts with relation **from** Kevin Bacon **to** Jodie Foster;
let S be **the next step via** the acts with relation **from** Christopher Walken **to** Kevin Bacon;

And of course, relations are integral to set-descriptions. Since set-descriptions can be parameter types in functions, procedures, rules, and understand tokens, and be a domain description in *repeat* loops and *now every* assignment statements, as well as passed-in and used in their own right to phrases like *filter*, they are the secret sauce which COBOL and Applescript lack.

repeat with countrymen running through **every resourceful not antagonistic person who trusts the player who is friends with a person (called the shill) who owns a thing (called riches)**:
say "Hi [countrymen]. [Shill] said you'd lend me your [riches]."

In the above description, each subordinate phrase applies to the main noun -- person, in this case -- not to the nouns listed in other subordinate phrases -- such as player, thing, or the second person. (Asking "who is friends with a person" is a way of asking who has friends.) We cannot insert commas or conjunctions ("and").

Rules of Thumb

Because interactive fictions are single-author artworks rather than team-designed workhorses, novelty and ease of modification trump safety and scalability. As a result, a partially rule-based paradigm was chosen. Where the procedural paradigm triggers imperative code blocks by invoking that code's given name (the name-plus-code construction is called a procedure or function), the rule-based paradigm triggers imperative code blocks by attaching situational information to it. The situational condition is called the preamble, and the preamble-plus-code construction is called a rule.

In other words, a rule encapsulates *when* its code executes. For a procedure, the information about when it executes is spread diffusely over the source code: wherever its name appears. Rules typically don't need names because rules already know when they execute. They needn't be told so by other code. (Rules can be named, as the following example shows, and we'll soon see some reasons for doing so.)

> A carry out rule for someone playful (called the prankster) which is friends with the player inserting a favorite something into something worn when the time of day is before the deadline during the opening scene **(this is the** my fairly long **rule):**
>
> now the prankster annoys the player.

We have already dissected two rules so far in this guide, one in the nutshell chapter and one in the firehose chapter, so let's move a little faster through this one. *Carry out* would be the rulebook name, except that *check*, *carry out*, and *report* are actually each multiple rulebooks, one set for each action. So the rulebook in use here might be named *carry out inserting it into*. *A* and *rule for* are just syntactic sugar, and ignored by the compiler. The first parameter, the performer, must match the set-description *someone playful which is friends with the player*, and if so, the variable *prankster* captures. *The noun* will hold what matched *a favorite thing* while *the second noun* will hold what matched *something worn*. The *when* condition must be satisfied and a scene, *the opening scene*, must be currently in progress. We also name this rule *my fairly long rule*, so further work may be done on this rule, such as temporarily ignoring it at run-time via an extension, or placing constraints about its placement in its rulebook vis-a-vis the other rules in it. The colon begins the imperative code block, and the period ends the imperative code block. In short, the rule defines in what situation that the player will be annoyed by someone.

In Prolog the rule would be written the other way around: *The prankster annoys the player if-and-only-if the action is the inserting it into action, and the actor is a person, and the actor is playful, and the actor is friends with the player, and the noun is a thing, and the noun is a favorite, and...* and so on, in its verbose, obtuse, punctuation-filled way. Neither COBOL nor Applescript are even that sophisticated. Indeed, one of the better analogs for Inform's rules are the rules of Perl 6, even though Perl's rules are intended primarily for text strings and grammar.

A few short examples of rule preambles follow. The latter three have the imperative block included as well.

A persuasion rule:
Persuasion: [an identical preamble to the above]
Every turn:
Every turn during the collapsing bridge scene:
Report someone burning something:
Instead of burning something held (this is the safety first rule): say "No, you might burn yourself!"
Carry out **an actor** helping someone: now the noun befriends the actor. [*an actor* means anyone; *an* is required]
Report tattooing: say "You go to work on [the second noun]'s [the limb understood]."

Unlike Prolog's never-ending sea of rules, Inform groups its rules into rulebooks. When a rulebook is invoked, only that rulebook's rules are considered. The only required part of a rule's header is the rulebook to which it belongs, though additional clauses are frequently attached using *during* (for scenes), *when / while* (for *if* conditions), and a sole action description (such as *someone cutting something*) immediately following the rulebook name. This allows rulebooks to encapsulate smart, mutable behavior, similar to objects, but behavior here is paramount, not a data instantiation. It could be said that objects implement nouns while rulebooks implement verbs.

In source code, a rulebook's rules are rarely found together or in any particular order. They can be scattered all over the source code, sprinkled throughout extensions, etc. An author is free to group his rules however he wishes, such as by the narrative scene or geographical area in which they're used, or by the rulebook to which they belong, or, yes, by the object(s) to which they apply.

There are a few cases where naming a rule is useful, but these are usually in addition to the header, not in spite of it. One is in debugging output: the RULES and RULES ALL testing commands list the names of rules as they execute, or are considering execution, respectively. Two is the ability to put the rule into a rule variable or passing the rule to a phrase. Three, rarely, is imperatively invoking a particular rule. Four, especially for extension authors, is so other rules may be listed before, after, or instead of the named rule at compile time. Finally is the sake of documentation, primarily the index.

The block sleeping rule **is not listed in** *any rulebook*.
My magic hand rule **is listed instead of** the can't take distant objects rule **in** the check taking rulebook.
The landlubbers can't walk right rule **is listed first in** the check going rules.
This is the my foobar **rule:** say “Someone please invoke me imperatively or list me *instead of* another rule!”

But rules aren't functions. The latter example names a rule but gives it no preamble and no containing rulebook. Unless *my foobar rule* is itself used in a line like the preceding two, or is directly invoked with *follow*, it's dead code. The named but preamble-less rule implies we are thinking too procedurally. Whatever invokes that rule is itself invoked at a particular time under particular circumstances, so those circumstances should be in the preamble. Even if the rule is invoked from multiple places, at least one place can use the preamble.

Strictly speaking, a rule accepts only one parameter, as defined by its rulebook, while all else

in the rule's preamble comes from global variables – the world-state. (Even more strictly: at the Inform 6 level, the parameter itself is actually in the global variable *parameter_object* and rules become functions taking zero parameters.) However, the default type of a rulebook is *a description of actions,* which itself accepts at most three noun parameters, each of which could be a set-description. Since most of the commonest rulebooks do this, rules seem to have a higher arity than they really do. Furthermore, a rule can optionally return one value, via *rule succeeds with result*, the type of which is defined by its rulebook. All of the standard rulebooks produce nothing, which further muddies the true nature of rules: as a kind of codec, filter, or transformation which accepts the world-state in a particular state, possibly prints something, and possibly changes the world-state as it finishes.

So to make a long story short: when a rulebook is *K based*, then rules belonging to it are written *rulebook-name instance-of-K*, and even then the instance is optional.

Snark is a **text based** rulebook.
Snark "your mother was a hamster": [...]
Snark "silly English k-nigget" during the siege of the French castle: [...]
Last snark: […]
The last snark rule: [...]

Spell out is a **number based** rulebook.
Spell out 15: [...]
Spell out:
Spell out 2 during relationship counseling: […]

Juggle is an **object based** rulebook.
Juggle the hamster: […]
Juggle an animal when PETA is not in the location: […]
The first juggle something (called the whatsit) when the player does not carry the whatsit: [...]

Ms Manners approves is a rulebook. [basis is missing, so *a description of actions* it shall be]
Ms Manners approves someone nice wearing something nice: […]

So let's examine rulebooks now, as they drive what a rule can, should, and cannot do, including defining the apparent syntax even if the grammar is identical in all cases.

Rulebooks: White-box Paradigm

Like a function, a rulebook solves what task is given it. Unlike a function, a rulebook picks and chooses which rules within itself to execute. It will execute its rules until one of them produces a success or failure result, distinct from any (optional) returned value. In lieu of a return statement, rules use one of the *rule succeeds [with result X]*, *rule fails*, or *make no decision* imperatives, the last of which tells the 'book to try the next rule for a result. If a rule ends without one of these three imperatives, the default imperative -- usually *make no decision* -- is invisibly added at the end of every rule. That default is set by the rulebook.

```
The pick a plan rules are a rulebook.
The pick a plan rules have default outcome success.
[ Or failure, or no outcome ]
A pick a plan rule: say "I always fail, regardless the rulebook's default."; rule fails.
A pick a plan rule: say "I can never make up my mind so one of my peers will now
execute."; make no decision.
A pick a plan rule: say "I exhibit the default behavior for the rulebook."
A pick a plan rule: say "I'm always a winner."; rule succeeds.
```

A rulebook can take up to one parameter (including one whole action-description) and *produce* one return value. A rulebook is invoked in one of several ways, mainly depending on whether it takes a parameter and whether it produces a value. These phrases are 1) *follow (rulebook);* 2) *follow (rulebook) for (value);* 3) *the (value) produced by (rulebook);* and 4) *the (value) produced by (rulebook) for (value).*

A snarky one-liner is **a *rule* based rulebook producing *text***. [*rulebook* by itself is *action based producing nothing*]

When play ends, say **the *text* produced by *snarky one-liner* for *the reason the action failed*.**

Snarky one-liner for the shouldn't touch electrical wires rule:
rule succeeds with result “You ignored the No Trespassing sign? I'm... shocked.”

Snarky one-liner for the shouldn't descend deep dark wells rule:
rule succeeds with result “Good news! You have a choice: drowning or landing.”

The remaining ways all substitute *follow* for either *abide by* or *anonymously abide by*. Both of these return immediately after their called rulebook returns. The only difference is the latter not setting the rule variable *the reason the action failed* to itself, becoming a silent middle-man. Finally, *rule succeeded* and *rule failed*, the past tense versions of *rule succeeds* and *rule fails*, are used in *if* statements to see if their present tense counterparts ended the recently called rulebook. This information is stored just above the current stack pointer, so calling any functions – including things which become functions in Inform 6, such as set-descriptions – will clobber them.

```
Check planning:
    follow the pick a plan rules;
    if the rule succeeded:
        say "Rulebook succeeded.";
    otherwise if the rule failed:
        say "I've no plans because of [the reason the action failed]." instead;
    otherwise:
        say “Not a single applicable rule could make up its mind.”

Check planning (this is the choose-a-plan rule):
    abide by the pick a plan rules;
    say "This only prints if the pick a plan rule make no decision.  But if it did
    succeed or fail, abide by returns, setting the reason the action failed to this
    choose-a-plan rule."

Check planning (this is the choose-a-plan rule):
    anonymously abide by the pick a plan rules;
    say "As above, but the reason the action failed will still be set to a rule in the
    pick a plan rulebook even upon success and failure.  This choose-a-plan rule is
    a silent middle-man."
```

One way to think of rulebooks is as callbacks. More restricted languages, especially C when talking to an operating system's API, use what are called callbacks to “insert” user-written code into an operating system or library. Typically, callbacks are a bit arcane. C is notorious for tangling pointers like spaghetti, so pointers-to-functions – which is what callbacks are – can be a recipe for cyanide. But Inform's problem domain of iconoclastic games, which thrive at least in part on novelty, places heavy demands on the flexibility of any standard library. Callbacks are the norm, not the exception. Whence comes rulebooks: “open” spaces where rules may be inserted and/or swapped out. So instead of code resembling this:

```
The callback rules are a rulebook.  [usually empty]

To foobar:
    ponder foo;
    follow the callback rules;
    mull over bar.
```

... the pattern becomes this:

```
The foobar rules are a rulebook.
Foobar rule (this is the pre-callback work rule): ponder foo.
Foobar rule (this is the post-callback work rule): mull over bar.
```

And now client code may insert code in a variety of places, even replacing the imperative sections if desired.

To further strengthen the tie between rulebooks, functions, and natural language, rulebooks may have “named outcomes”. These are a set of imperative phrases and a set of named values

which work in concert. The outcome identifier by itself is the imperative, which is syntactic sugar for *rule succeeds with result <corresponding named value>*. The named value set each have the word *outcome* appended (except when printing) and fill-in the variable *the outcome of the rulebook* for use in the caller's code. Outcomes can be classified as success or failure as well.

```
The audible rules have outcomes silent (failure), whispered (success), voiced (success
- the default), and deafening.

[...]; whispered.     [ synonymous with rule succeeds with result the whispered
outcome ]

follow the audible rules;
if the outcome of the rulebook is not the silent outcome, say "You heard something
[outcome of the rulebook]."
```

Named outcomes are most useful for extensions and the standard library. Much as boolean object properties are a pair of named values (like *lit/unlit*) belonging to an anonymous property, rather than a named property (like isLighted) holding true or false, rulebooks can provide easily remembered decision phrases for new, author-written rules.

```
Visibility when the player carries at least two lit candles: there is sufficient light.
Does the player mean taking the train: it is very likely.    [ "getting on" or "entering"
the train, perhaps ]
For deciding whether all includes the kitchen sink: it does not.
For reaching inside a closed tiger cage: allow access.       [ foolish, but possible ]
```

Rulebooks may also have local variables. Unlike *let* variables which end with the code block, these remain in scope over all the rules of the 'book. Though we don't have a specific way of initializing them like *let* does, we can use the rule ordinal *first* to put a particular rule at the start of a rulebook. (Only *first* and *last* exist this way, and in cases of multiple rules using them, only the final *first* or *last* rule actually is so, superseding the preferences of earlier rules that use them.) *Last* rules are useful for providing a fallback.

```
The pick a plan rulebook has a rule called the best plan.
The first pick a plan rule (this is the initialize plan rule): now the best plan is the little-
used do nothing rule.
The last pick a plan rule (this is the no idea rule): rule fails.
```

A few odds and ends before we close the chapter. *Stop the action* is synonymous with *rule fails*, and *continue the action* is synonymous with *make no decision.* The word *rules* is synonymous with *rulebook* in the source code, and is possibly the only case in the language where a singular (*rule*) and its plural (*rules*) do not mean the same thing. (In practice, this exception is fairly natural.) Finally, a rule variable can hold a rule and/or rulebook, but a rulebook variable can only hold a rulebook.

Events are Actions

In interactive fiction, the player controls the protagonist, and typed commands translate into actions within the fictional universe. Protagonist actions like *looking*, *examining*, or *burning* have rulebooks to deal with those events. Specifically, three rulebooks comprise an action: its *check* rulebook ensures the action can proceed; its *carry out* rulebook updates game state; and its *report* rulebook narrates the result. The other rulebooks that hinge on an action, such as *instead*, these three are created anew for each action in the game. There isn't one *check* rulebook; there are many, one for each action. This is done partly for performance reasons, and partly for sanity reasons: to avoid a "sea of rules".

```
Check taking something held: say "You already have that." instead.
Carry out an actor taking something: now the actor carries the noun.
Report taking: say "Taken." instead.
```

Much of Inform programming is writing report rules for actions, though some game events like *every turn*, *printing the name*, and *when play begins* are also popular. (Game events usually comprise a single rulebook apiece unless they are *activities*, which we will see shortly.)

Defining a new action is similar to a function prototype. The exact wording *is an action* is required. Furthermore, at least one "parameter" must be given: *whining is an action* is disallowed without some more information. However, Inform doesn't much care what that information is: supplying a past participle is just as good as the *applying* arity and type specification. *Applying to nothing* is assumed if it is dropped.

```
Donating is an action applying to one thing.
Discussing is an action applying to one topic.
Accusing it of is an action applying to one object and one visible object.
Tabulating is an action applying to one number and requiring light.
Scheduling is an action applying to one time.
Temporarily waiting is an action applying to one time.
Whining is an action applying to nothing.
Teleporting to is an action applying to one object.      [room, to be specific]
Debugging some stuff is an action out of world applying to nothing.
Tattooing it of is an action applying to one limb and one thing, requiring light.
Weaving is an action with past participle woven, applying to one thing.
```

These each create the three rulebooks that will hold the implementation details. Because actions implement an English verb that our player-character (or an NPC) will perform, verbs have a maximum arity of three and a minimum arity of one. The subject, called *the actor* (or, as a global, *the person asked*), is required so no mention is made. In the implementation, the direct and indirect objects of a sentence are called *the noun* and then *the second noun* – but only if they are objects. Otherwise, the variable is *the number understood*, *the time understood*, *the topic understood*, etc.

The syntax for declaring actions is very special-case. The phrase *requiring light* refers to a

worldsim precondition. The adjective *visible* places every object (such as intangible ideas and rumors) in scope, and is completely unrelated to *requiring light.* The word *thing* (or *things*) actually means any objects – we can't be more specific with *person*, for example. (However, it does accept *object* as a synonym, oddly enough.) The phrase *out of world* is for system-level actions, which skips a great deal of the worldsim's calculations. *With past participle X* tells Inform to use that word in source code for the past participle (*has woven* for example) instead of replacing the -ing suffix with an -ed suffix automatically. And finally, other types such as topic, time, time period, number, and various units may each only be used once in an action.

After an action is created, the action must be implemented with rules, and then the parser must be pointed to it via Understand statements. If the parser is not pointed at it, an action can still be invoked from the code, via *try*, *try silently*, and *silently try*. When tried *silently*, the *report* rules are skipped entirely.

```
    try Bob donating the jeans;
    silently try donating the red Ferrari;
    try Bob accusing the player of theft;
    try teleporting to the tattoo parlor;                    [subject can be dropped when
    "the player" performs the action]
    try silently the current manager tattooing the back of the player;
```

For actions with two objects, at least one word (such as a preposition) is needed between the parameters, and the word *it* is a placeholder for the first parameter. *It* tells Inform where the first parameter goes, because actions with multi-part names like *teleporting to it by way of* have too many choices otherwise.

Finally, an action invocation can be put into a variable of type *stored action* and invoked at a later time. The phrase *the action of* must precede the invocation in order to capture it.

```
    An abeyance is a stored action that varies.

    [...]; now the abeyance is the action of Bob examining the player;
    [...]; try the abeyance;
    [...]; now abeyance is the action of the current manager firing the noun;
```

The parts of the stored action can be reached with syntax similar to object properties: *the actor part of, the action name part of, the noun part of,* and *the second noun part of*. Stored actions are easy to create when only the nouns vary: just use variables like *the current manager*. But to assign to the *action name* part the extension Editable Stored Actions is necessary. A built-in *to decide* function, *the current action*, returns the currently-executing action as a stored action. (Syntax note: very occasionally, the compiler will have difficulty parsing an invocation that has a NPC as a subject. In these cases, add the word *trying* before the action: *Bob trying examining the player*.)

Processing a character's action is a multi-step process, not including parsing. Each step is its own rulebook. And just to be clear, every action has its own *check*, *carry out*, and *report* rulebook. The others are shared among all actions. The following outline shows the rulebooks chronologically, indenting the NPC-only rulebooks.

Setting Action Variables
Before
Persuasion
[NPCs only]
Instead [default outcome is failure, so only one rule will typically execute before jumping to the end]
Check <action>
Unsuccessful Attempt By [only when Instead or Check fails] [NPCs only]
Carry out <action>
After [default outcome is success, so only one rule will typically execute before jumping to the end]
Report <action> [skipped if the action was invoked with *try silently*]

Each rulebook has its task:

Check rulebooks catch situations which should prevent an action from occurring. It enforces preconditions, essentially. If the check rules prevent the player from doing something, it is responsible for narrating that result. So a typical check rule reads like, "if X, then say Y and abort action."

Carry out rulebooks simulate the action, updating any data and running any code necessary. It should not print anything lest *try silently* won't live up to its name.

Carry out an actor jumping: now the actor is on the nearby platform. ["an actor" applies to everyone]

Report rulebooks then narrate the result of the successful action. Usually the say statements in a report rule should end with *instead* to prevent multiple report rules from running. The action is already considered to have succeeded by this point, so the usual meaning of *instead* as failure isn't significant.

Report someone jumping: say "You see [the actor] jump over." ["someone" applies to any NPC]
Report jumping: say "You jump over." [subject-less means the player]

Persuasion decides, upon a player command like BOB, TAKE ROCK, whether Bob will do as you ask him to. The default *persuasion* rule, written in Inform 6 and so omitted from the Actions index, denies with "Bob has better things to do."

Unsuccessful attempt by runs if Bob agreed to what was asked of him, but a *check* or *instead* rule stopped him. It exists so the author may narrate failure better than the default, "Bob is unable to do that." Again, the default rule is written in Inform 6, so is omitted from the Actions index.

Before happens between parsing and game reaction. It can also trigger on groups of actions. It's useful for setting up variables or other miscellaneous situations before the action rules

begin in earnest.

Instead is useful for blocking groups of actions. This is a favorite of most new authors, and it's not hard to see why. It can apply to groups of actions or a single action, one-liner versions can use the comma for great readability, it defaults to *rule fails* so writing a rule here can handle an entire situation on its own if desired. Unfortunately, *instead* rules tend to get used for everything, which is a bad coding practice because there are knock-on effects later. For example, the system thinks every rule fails, so tensed conditions like *if we have examined the red book* will fail, even though we have, in fact done so. The author's prose came from an *instead* rule instead of a *report* rule.

> Instead of Tiny jumping: say "Tiny is too overweight to jump. You all must find another way to help him across."

After, when adorned with a condition or three, is useful for triggering cutscenes. It defaults to *rule succeeds*, so the report rules aren't run afterward. Again, they tend to get used often for the same reasons as *instead*.

Setting action variables should do as little as possible, and is rarely needed. It's intended to initialize variables that are local to an action, as opposed to variables local to an individual rulebook or rule, by pulling values from global variables and object properties, including the parameters *the actor*, *the noun* and *the second noun*. When declaring an action-local variable, an optional "matched as" parenthetical allows rules in the rest of this action's gauntlet to test those variables similarly to how it tests on the actual parameters. It isn't a way to pass additional parameters to a rule, only to check world-state readably. The standard *going* action uses this to provide rule hooks like *going to/from <a room>, going through <a door>, going by <a vehicle>*, etc.

> Report **going from *the monkey village*** **to *the lost city's entrance*** **by *the hoverboard***: say "You sail into the clearing easily this time, bypassing all those nasty monkeys that like to drop on top of you." instead.
>
> The jumping action **has** an object **called** the obstacle **(matched as** "over"**).** [*Matched as* must be a single word.]
>
> **Setting action variables** for jumping: now the obstacle is the current blockage of the location.
> [This pulls data from a "current blockage" property of a room, specifically, the room the player's currently in.]
>
> Instead of Stevie Burns jumping: say "Even little Stevie hops over [the obstacle]."
> [Jumping takes no parameters beyond the performing actor]
>
> Instead of Stevie Burns jumping **over** a fiery fallen beam: say "'Help!' says Stevie. You suddenly recall him telling you about the time his house burned down." [We can now test on the local variable with an "over" phrase.]

Because the *before*, *after*, and *instead* rulebooks are shared among all actions, they may apply to entire categories of actions. A named group of actions is called a *kind of action*, and they

have no special declaration syntax beyond the word *is* sitting between the action's name and the kind-of-action's name. This encourages all manner of nouns, adjectives, and adverbs to name a kind-of-action, but some read better than others when used in rule headers. In all cases, an action must be defined before attempting to classify it.

```
Whining is pointless behavior.                [ "pointless behavior" is now a
kind of action ]
Temporarily waiting is pointless behavior.
Discussing is conversation.
Accusing it of is conversation.
Accusing it of is drama.
Teleporting to is drama.
Tabulating is acting like a frickin' accountant.
Scheduling is acting like a frickin' accountant.
```

Kinds of actions can then be used in rule preambles. In the following examples, remember than *when* and *during* begin additional clauses that check global variables and whatnot.

```
After pointless behavior: say "But you still feel unfulfilled."
Before conversation when the current interlocutor is not here, now the current
interlocutor is a random person here.
Instead of drama, say "(Now is the time to lay low!)"
Instead of acting like a frickin' accountant during the collapsing bridge scene, say
"You calculate (correctly) that you're about to become a victim of natural selection."
```

Any rulebook that can accept a kind-of-action can accept an explicit list of actions as well. Let's then call that list Feature #1, because of the many variations on it. There is a special kind-of-action, the noun phrase *doing something/anything*, that matches all actions. (Feature #2). Optionally, it may [#3] be followed by an *except* or *other than* clause that lists actions. Then, optionally, we may [4] tack on a set-description to further constrain the rule. (The set-description can [5] be preceded with *with* or *to* if we wish because it frequently improves readability.) And of course, we may [6] always add other conditions with *when* and *during*, such as the *in the presence of <person>* boolean function.

```
Before an actor discussing [4] a spiffy thing [6] when in the presence of Mr
Blackheart: [...].

Instead of [2] doing anything [3] except waiting [6] when the player is paralyzed, say
"(Uhh... can't... move...)"

Instead of someone [2] doing anything [3] except taking, dropping, or burning [5] with
[4] something incriminating, say "[The actor] says, 'No, I must get rid of [the noun]!'"

After [1] examining, looking under, or searching [4] anything owned by Mr Blackheart
[6] during a scene needing tension: say "Suddenly, Blackheart re-enters the room.
'What are you doing.' It wasn't a question."
```

Just to be clear, *doing something* is a single identifier. *Doing* is not an action, and its following *something* is not a set-description parameter. But *something* is a set-description

elsewhere, as in *examining something* or *accusing something of something*.

There's six caveats to know. One, when using *doing something*, we must remember that *looking* is what primarily prints text when we enter a place – or re-prints it when we load the game, etc. – so we frequently will want to allow it. Two, when using *doing something to/with/-- something*, actions of low arity like *sleeping* and *waiting* won't be caught by it, because they never apply *to/with something*. Three, the docs don't cover any syntax for constraints on *the second noun*, but a *when* clause can be appended. Four, when explicitly listing several actions together, they must *check* the same arity. (So we can't combine *sleeping* with *examining something*, but we can combine *sleeping* and *examining*. The arity of the check, not of the action, is important.) Five, but we *can* combine them using a *kind of action* (see immediately below). And six, in all cases, the subject of the actions is kept completely separate from the rest of the action-description. Consider it already processed by the time the verb is reached.

```
Examining something is acting like a klutz.  [ not "an actor examining something" ]
Dropping someone is acting like a klutz.  [ notice the "someone" ]
Looking is acting like a klutz.  [ arity of the check is lesser than the other two ]

Before someone acting like a klutz: [...].  [works for any NPC examining a Thing, dropping a Person, or Looking]
```

Understanding Our Player, Our Parser

The player's parser is a fairly simplistic one, little changed over a decade of use, but it makes heavy use of callbacks for some surprising extensibility. These parser callbacks are called topics (meaning, they are of type Topic). Topics are simplistic regexes in the shape of a boolean function, and if they match, they "return" what object, unit, action, or value they find by setting global variables. A given topic must "return" only one particular type; more on that in a moment. We use Understand sentences to add these topics to the pre-existing list of other topics that the parser runs down. Here are some examples of using Understand to connect topics with actions:

Understand "whine" **as** whining.
Understand "donate [something]" **as** donating.
Understand "give away [something]" **as** donating. [We're making a synonym here.]
Understand "discuss [text]" or "talk about [text]" **as** discussing. [Ditto, but as one topic not two.]
Understand "tabulate [a number]" **as** tabulating.
Understand "answer [truth state]" **as** answering.
Understand "schedule [a time]" **as** scheduling.
Understand "wait for [a time period]" **as** temporarily waiting.
Understand "teleport to/-- [any room]" **as** teleporting to. [The altercation slash can't be used on the first word]
Understand "accuse [someone] of committing/-- [any thing]" **as** accusing it of.
Understand "[any thing] committed by/via [someone]" **as** accusing it of **(with nouns reversed)**.
Understand "wear [something preferably held]" **as** wearing.
Understand "put [other things] in/inside/into [something]" **as** inserting it into.
Understand "deposit [something] in/into [an open container]" **as** inserting it into.
Understand "go to [any adjacent visited room]" **as** going by name.
Understand "tattoo [limb] of [someone]" or "tattoo [specific limb] of [someone]" **as** tattooing it of.

The whole text between *understand* and *as* is a topic, and the text in the square brackets is yet another topic called by the larger topic. The called topics start their parsing where the calling topic left off, trying to decide if the words that follow match itself.

The topic *any thing* – as opposed to *thing* or *a thing* or *something* – ignores the limitations of scope and will match any valid object found in the whole game. The word *any* in general works like this, and must be echoed in the action definition by the word *visible*. Other topics are intended for a particular type – *time, number,* a particular *unit*, named values like *limb*, undigested *text* which actions reference by *applying to one topic*, etc. We may also use set-descriptions here: *an open container*, *any adjacent visited room*, *something related by reversed containment,* and so on.

Understand can also expose other parts of our games to our players: property adjectives, synonyms for object names, and even other topics.

Understand "dog" **as** Rover.
Understand "birds" and "ruddy ducks" **as the plural of** duck.
Understand "upper [limb]" or "lower [limb]" **as** "[specific limb]". [this won't capture the words Upper or Lower, but allows the player to use them]
Understand "beneath/under/by/near/beside/alongside/against" or "next to" or "in front of" **as** "[nearby]". [this is a convenience for the programmer only]

Any single topic can only return one specific type of something.

Understand "colour [a colour]" or "[something]" **as** "[tint]". [ERROR: is this Topic's return type *color* or *thing*?]

A pot is a kind of thing. A pot can be broken or unbroken.
Understand the unbroken **property as referring to** the pot. [or, "...the broken property..."]
Understand "shattered" or "cracked" or "smashed" **as** broken.
Understand "pristine" **as** unbroken.

Understand the broken **property as describing** a flowerpot. [describing disallows "take broken"; player must provide the noun: "taken broken pot"]

We can understand kinds (classes) which will at least narrow down the player's possible referent.

Understand "machine" **as** a device.
Understand "bottle of [something related by containment]" **as** a bottle.

If conditions can be attached to any *understand* statement, so the parser will ignore the line if its condition isn't met.

Understand "rouge" **as** red **when** the make-up set is visible.
Understand "Rover" **as** Rover The Dog **when** the player knows-about Rover.
Understand "your" **as** a thing **when** the person asked has the item described.

But remember that this happens during parsing of the player's command, so the condition can't always refer to information within the command itself, such as *the noun* or *the second noun*. In the case of *the person asked*, it works for FRODO, GIVE ME YOUR RING because Frodo has been parsed already, but will likely fail if "your" is the first word of the command.

If we just want a simple reply with no processing, there's a shorthand.

Understand "xyzzy" **as a mistake (**"Ah, I see you're an old hand at this."**).**
Understand "xyzzy" **as a mistake (**"The machine doesn't seem to have a button with that label on it."**) when** in the teleportation chamber.

Mistakes can use the *[text]* token but little else. They can use *say-phrases* in what they print, so it is possible to do some simple codework from them. Mistake lines are parsed before non-mistake lines, because they're intended to cover exceptions to grammar or one-off, out-of-

game remarks by the player, rather than entire categories of input.

Finally, a distinction in the *time* type that understand makes but the rest of the compiler does not. The *time* datatype comes in two flavors, instants and durations. Three and a half hours would be a duration, while 3:30 pm is an instant, a specific point on the timeline. Internally, both are stored as the number of elapsed minutes since 4 am, but actions need to know which flavor the player types in. So, only *understand* assertions distinguish between, *[time],* an instant, and *[time period],* a duration. Math is straightforward once after parsing: adding or subtracting a duration produces whichever it's being combined with, while subtracting two instants produces a duration. But adding two instants is nonsense: what could 9:38 pm plus 4:47 am possibly be?

```
Understand "wait until [a time]" as waiting until.          [ wait until a moment,
such as 1 pm ]
Understand "wait for [a time period]" as waiting for.       [ wait for a duration, such
as an hour ]
```

Arrays Have Been Tabled

Inform is not a language of computation. It is a language designed to hold secrets for the player to discover. So its two-dimensional arrays – tables – are primarily intended for a single, occasional lookup. Tables are statically-allocated like everything else, and the words "row" and "column" are used in lieu of (x,y) coordinates.

Here's an example table from the docs:

Table 2.1 - Selected Elements

Element (some text)	Symbol (some text)	Atomic number (a number)	Atomic weight (a number)
"Hydrogen"	"H"	1	1
"Iron"	"Fe"	26	56
"Zinc"	"Zn"	30	65
"Uranium"	"U"	92	238

Or, alternately:

Table of Selected Elements

Element	Symbol	Atomic number	Atomic weight
text	text	number	number
"Hydrogen"	"H"	1	1
"Iron"	"Fe"	26	56

A blank line and the keyword *table* begin the whole affair. At least one tab character is required to end a column and start the next. Multiple tabs and spaces are OK. The first tab character Inform comes across melds together all whitespace surrounding it. Type information can either be left off – Inform will deduce it mostly correctly – or can be given in the two styles shown. Both styles have problems. The first, "inline" style only allows *object* for objects, not a particular class. The in-row style allows specific class names, but introduces a bug – row #1 will be a blank row that doesn't correctly flag itself as blank. This is because we can leave individual entries blank, which would need the type explicitly stated.

Table of Energy Proponents

Proponent	Fuel	Danger
Bob	Hydrogen	a number
Phoebe	Wind	--
--	Geothermal	--
Jean	Nuclear	10

with 4 blank rows.

A blank entry is denoted by the – double-dash. The last line (*with X blank rows*) statically allocates some extra space. A character's responses to the player's inquiries on various topics has special syntax, a column entitled *topic* whose type will also be *topic*. Remember that topics are simple regexes, so they cannot be printed.

Table of NPC Responses

topic	answer
"hi/hello"	"He says, 'Well hello there!'"
"bye/goodbye"	"'Take care,' he answers."

There's several phrases to work with tables, but it's still one of the more cumbersome parts of the language. This is because we must *choose* a particular row to work with each *(column name) entry* as if they were each a standalone variable. We can't pass a chosen row to another function. We can't choose two rows simultaneously, either in the same table or in different tables. We can't copy rows to other rows. Finally, we can't create N-dimensional arrays except by creating a column of type *table name* and the creation of a lot of statically-allocated sub-tables by hand!

But knowing this, the phrases used for tables is fairly self-explanatory. The following are all imperative procedures.

blank out (table entry)**;**
blank out the whole row;
blank out the whole (column) **of** (table)**;**
blank out the whole of (table)**;**

if (value) **is** (a column) **listed in** (table)**,**
if there is no (column) **entry,**
if there is (a column) **corresponding to** (a column) **of** (value) **in** (table)**,**
if there is (a column) **in row** (number) **of** (table)**,**

choose a blank row in (table)**;**
choose row (number) **in** (table)**;**
choose row with (a column) **of** (value) **in** (table)**;**

sort (table) **in** *reverse* (column) **order;**
sort (table) **in random order;**

These are functions.

the **number of blank rows in** (table)
the **number of filled rows in** (table)
the (column) **corresponding to** (a column) **of** (value) **in** (table)
the (column) **in row** (number) **of** (table)

And these are used in rule preambles, working somewhat like a set-description. Tables and topics are used together a lot, because it frequently happens a character has a whole array of things to say in response to various subjects. And the default types of the *asking it about* action are *person* and *topic*, respectively.

After taking (a column) **listed in** (table):
After asking someone about **a topic listed in** (table):

One concession to ease of use is that its sort routine is stable. Meaning, if multiple rows have the same value in the column we sort on, then their order vis-a-vis each other will not change.

A couple of features were added later to ease how tables and extensions interact. First is the *(continued)* parenthetical. If an extension creates a table, a game can append extra rows to it. Note that the order is important – a table section with *(continued)* must follow the table section without.

```
Table of NPC Responses (continued)
topic                    answer
"thank/thanks you/--"    "'You're very welcome.'"
```

The other feature is *(amended)* which allows the game to replace the pre-existing rows in an extension's table. Again, the amended section must come after the original, so order is important. Inform knows which rows are and aren't amended by looking at the left-most column(s) and matching them up. Generally, only simple values can be matched in this way. So the ordering of our columns actually matters as well, to the extension writer.

```
Table of NPC Responses (amended)
topic                    answer
"hi/hello"               "He says, 'Whaddaya
                         want?'"
```

Finally, when we use generic types, a *K valued table column* is the type of a typed column, mimicking *K valued property*.

Time for a Scene

The *room* class divides space into discrete places. *Scenes* divide an interactive fiction into durations of time. Each scene is implemented as a boolean variable that is automatically set and cleared by an asserted condition. Scenes happen only once, unless declared as *recurring*.

A lightsaber duel **is a scene**.
A lightsaber duel **begins when** the location of Luke is the location of Darth.
A lightsaber duel **ends when** Luke is too injured to continue or Darth is too injured to continue.

A person can be too injured to continue. A person is rarely too injured to continue.

Rule headers and imperative code both can check to see if a scene is happening.

Every turn **during** a lightsaber duel, say "BWWAAUUAAAHH".

[...]; if a lightsaber duel **is happening**, [...]

Each scene provides two rules that execute when the scene begins and ends. These are just hooks we may use or ignore.

When a lightsaber duel **begins**: now the command prompt is the battle command prompt.
When a lightsaber duel **ends**: now the command prompt is the normal command prompt.

Scenes are not objects, but may still have properties. Text floating out by itself after a scene definition goes into its *description* property, which is automatically printed when the scene begins.

The train leaves is a scene. “All aboard!”
A scene **can be** dramatic or dull. The train leaves is dramatic.
A scene **has** a person **called** its viewpoint character.

We can write set-descriptions about scenes as well.

Every turn during **a dull scene**: [...].

[...]; if **a thrilling scene** is happening, [...]

When **a thrilling scene** begins: […].

We can ask at what time, or how long ago, a scene began or ended with *the time since (scene) began, the time since (scene) ended, the time when (scene) began,* and *the time when (scene) ended.* Extensions provide further phrases.

Implementation-wise, scenes are just an example of named values, which follow. But because this is interactive fiction, scenes receive more syntax support from the language, as they should.

When the denouement ends, end the story finally.

Named Values Everywhere

Named values are a type, similar to *time* or *number* but finite in the number of instances. Inform uses named values everywhere. Scenes, rulebook outcomes, even a boolean object property is a named value of two values. We explicitly create new named values as a *kind of value*.

> A limb **is a kind of value**. The *limbs* **are** left leg, left arm, right leg, right arm, the neck, and the back.
> A tattoo **is a kind of value**. Some *tattoos* **are** mom, barbed wire, wings, a kanji, a pseudo-photograph, and some obscure symbol. Some *tattoos* **are** celtic knotwork and a tribal something-or-other.

Named values can have spaces in their names, but they cannot have their names abbreviated like objects can. Inform generally allows almost any word to be part of a named value, but be warned, if you put words like "when" or "is" in a named value, it may create compilation problems when used in other places. The articles used in defining a *kind of value* are usually ignored, but it is good practice to ensure both the singular (*a tattoo*) and plural (*some tattoos*) versions of the *kind of value* are used in order to prevent mystifying compiler errors later on. Finally, not all of a named value's names need be defined in a single sentence.

Named values can be used almost anywhere an object can. Plus there's a few phrases with named values can used but objects can't, since multiple object instances don't have any sort of ordering to them. In the following, the action *tattooing*, the relation *is/are tattooed with*, and the named value *tattoos* are all distinct from one another.

> now the offended body part is **the limb after** *the left leg*;
> now the offended body part is **the limb before** *the back*;
> now the offended body part is **the first value of** *limb*
> now the offended body part is **the last value of** *limb*
> now the drawn tattoo is **a random** *tattoo* **between** *wings* **and** *celtic knotwork*.
> **Definition:** a *limb* is hurt if [...].
> Tattooing is **an action applying to** one *limb* and one thing.
> **Understand** "tattoo [*limb*] of [someone]" **as** tattooing.
> Exposition **relates** various people **to** various *tattoos*. The verb to be tattooed with implies the exposition relation.
> Joan is tattooed with *some obscure symbol*.
> **repeat with** place **running through** each *limb*:
> A *tattoo* **can be** sleeved, stamped, or hidden. *Kanji* are *hidden*.
> A *tattoo* **has** a *topic* **called** the conversation starter.

Tables can defined a slew of named values. If used with the above, this method of definition must come first.

> Tattoos are a kind of value. Some tattoos **are defined by** *the table of designs*.
>
> Table of Designs

tattoo	*topic*
barbed wire	"barbed/wire/tattoo"
wings	"nice/angel/pretty"

Table of Designs (continued)

tattoo	*topic*
Jean-Pierre 4-ever	"who/was/jean/pierre"

Inform automatically creates some named values that correspond to other parts of the language. For example, *the containment relation* is an example of the relations named value, which corresponds to a built-in relation. This named value is passed into phrases such as the pathfinding and lookup functions mentioned in the relations chapter. New relations would be treated similarly: *the teenage love relation*. Likewise, *action name* is the named value counterparts for actions: *the looking action*, *the inserting it into action*, etc. Other times, one part of the language is little more than a named value.

It's Not Just Text

Though it's probably obvious by now, text goes between double quotes. Within it, square brackets denote a *say-phrase* (procedure call). Inform's print command is *say,* and it expands to a series of print statements and function calls in the generated Inform 6. As a nod toward the world of the printed word, a double quote within a string is written as an apostrophe, and the apostrophe itself is the say-phrase ['']. The say-phrase isn't needed in common words like don't or can't. As a convenience feature, a string ending in a period, question mark, or exclamation mark has an automatic line break appended. An extra space nullifies this behavior. So, in order to print:

```
Rene Descartes said, "I think not!" and promptly disappeared.
So don't ever say that, 'k?
```

… we use:

```
say "Rene Descartes said, '[']I think not![']' and promptly disappeared.[line break]So don't ever say that, ['][']k?";
```

Variables and objects in square brackets say their value or *printed name*, respectively. (The latter invokes the *printing the name* activity we'll soon see.) Other useful text properties are the *initial appearance* property, usually printed by the *looking* action, and the *description* property, which is usually printed by the *examining* action when it's on objects, or by the *when (scene) begins* rule for scenes. Some useful global text variables are *the command prompt*, which is usually the > greater-than symbol, *the left hand status line*, which is usually the name of the player's location, and *the right hand status line*, which typically holds the score. Communicating with the outside world are *the story title*, *the story author*, *the story headline*, *the story genre,* and *the story description,* which collectively create the “library card” used by websites like IFDB and in the game's own *banner text* seen at game start. Preview the library card in the contents tab of the index.

Inform was originally designed to make games with a very small memory footprint, so a game's output text is compressed and language support for regexes is poor. The *text* type is this compressed text. *Indexed text* is a later and still optional addition to the language. Regexes operate only on the latter but can implicitly typecast the former. Both types use double quotes identically, so constructions like the following are needed to avoid creating plain *text*.

```
let T be indexed text;
let T be "Hello World";
```

Manipulating the player's inputted text is mediated through *snippets*, *topics*, and the phrases *if (snippet) includes (topic), if (snippet) matches (topic),* the *does not* negated versions, *cut (snippet),* and *replace (snippet) with (text)*. A *snippet* is a pair of numbers, joined together like fixed-point notation, that represent a range of words. "The 3 words starting at word #2" would be the snippet 203. There are only three built-in *snippet* variables: *the player's command*, *the matched text*, and the deviously-named *the topic understood* which is used in actions *applying to one topic*.

After reading a command:
    if *the player's command* **does not include** *"please/thanks"*, say "How rude!" instead;
    if *the player's command* **includes** *"please/thanks"*, **cut** *the matched text*;
    if *the player's command* **matches** *"hi there"*, **replace** *the matched text* **with** "hello";

Report asking Bob about *"home/leaving"*:
    say "Bob rambles on about how only slackers think of [*the topic understood*] so much." instead.

Understand *"bastard", "rat/prissy bastard", or "sir"* as Bob.

Topics are regexes from a time when the term "regular expressions" was still mathematically correct. They are written within double-quotes and depend upon the invoking phrase to infer the type. Their only operator is "or", indicated by the / forward slash, or, in *understand* lines, by commas and conjunctions when outside quotes. Topics are frequently used in a game's conversation system because many subjects aren't physical objects which most standard actions need manipulate, and regexes require less memory than objects. (Incidentally, this is also why the object hierarchy lacks an *idea* or *concept* class.)

Indexed text is troublesome because Inform tries to remain statically-allocated, but the full Perl-like regex system works on it. Typecast between snippets and indexed text like so.

let the command-to-be **be** *indexed text*;
let the command-to-be **be** the player's command;
[...];
change the text of the player's command to the command-to-be; [this is a special-case syntax]

Precisely One Spoon-unit Of Sugar

Units are user-defined numeric types with some nifty syntactic sugar, sugar that extends even to our player. Generally in programming, numbers are numbers: the programmer must remember one variable measures in pixels while another measures in picos, and to mix the two makes bad medicine. But Inform allows us to define unique ways of writing the numbers -- by wrapping identifiers and punctuation marks (the "preamble") around them. The individual numbers ("parts") in the construction can be given names for later referencing.

```
Money is a kind of value. $19.99 specifies some money with parts dollars (without
leading zeros) and cents (optional, preamble optional).
My wallet is some money that varies. My wallet is usually $20.75.
A thing has some money called the price. The price of a thing is usually $5.
The price of Bob is $2.05.

say "[the dollars part of the price of Bob]";
```

We can only add and subtract units, assuming the units match. To allow multiplication and division, we can specify what new unit is created. Note the re-use of the specify/specifies verb.

```
A length is a kind of value. 10 m specifies a length.
An area is a kind of value. 10 sq m specifies an area.
A length times a length specifies an area.
```

We can always multiply and divide a unit by a plain number. Multiplying and dividing by one is how we explicitly typecast between units and numbers. We can also define units that are scaled versions of each other. Inform doesn't support floating point, but it can combine the types into a fixed-point type this way.

```
1m specifies a length. 1cm specifies a length scaled down by 100.
1m specifies a length scaled at 100.            [ This has similar effect without defining
cm. ]
```

Inform automatically creates a new Understand token from units. The player writes the unit the same way we do.

```
Understand "donate [money]" as donating. Donating is an action applying to some
money.
```

Finally, we can re-use the *implies* statement from defining relations for an extra dose of sugar.

```
The verb to cost (it costs, they cost, it is costing) implies the price property.
The jeans cost $19.95.
```

Note that we didn't define a relation there, though it may look like it. But we could:

Fanciness relates a thing (called X) to some money (called Y) when the price of X > Y.
The verb to be fancier than implies the fanciness relation.

let L be the list of things fancier than $2.50;
let B be the list of things fancier than the price of jeans;

The built-in type *time* is not a unit, but pretends to be. Time's parts – hours and minutes – are calculated from a 4 AM minute-count, not stored as separate numbers as units are done.

Inform also supports dimensional analysis on units when used in *equations*. Mathematics is the ultimate declarative language: it needn't compile to assembly. So Inform sets off its declarative equations from the normal flow of code and prose. Variables are one to ten letters long, and may be set right next to each other to represent multiplication. Whitespace is insignificant. The types of the variables are listed afterward in a *where* line, always number or a particular unit, and possibly a global variable or exact value.

Equation - Volume of a Cylinder
    V=pir^2h
where V is a number, pi is 3, r is the radius, h is a number.

The radius is a number that varies.

The local variables can appear outside the equation, especially when we use the equation to find that value.

let h be 15;
let V be given by the volume of a cylinder, where r is 5;

The values for all variables except one must be suppled by the time *given by* is invoked. There's four ways to supply a value. One is as a constant, as we've done with *pi*. Two is by global variable, as we've intended with *radius*. (Note the compiler won't ensure we've initialized the global.) Three is by declaring and initializing the variable with *let* like any local variable, as we've done with *h*. And finally is the *where* line itself, as we've done with radius (*where r is 5*) and overridden the global. The answer will either go into the local we declare with *given by*, as we've done with *V*, or into the global it is tied to, such as the following which populates the global *radius* but leaves *r* undefined.

let r be given by the volume of a cylinder, where V is 25, and h is 2;

Note that the above knew to use the square root to find the radius, but we cannot specify the square root in the equation ourselves. Besides parentheses, equations support addition, subtraction, multiplication, division, and exponentiation directly, and for squares and cubes, can rearrange terms to take the square root or cube root as appropriate. But we cannot specify roots in equations ourselves, so we would have to code the quadratic equation traditionally.

Even with those constraints, there are many equations that Inform cannot rearrange. Polynomials are a fine example. And all math here is integer – Inform does not currently support floating point. (Fixed point is supported, but only as part of a *scaled* unit.) The

equations feature is not intended as a mathematical tool, but as a coding feature for convenience and correctness. If we need to solve for multiple different variables from the same formula, it is better coding practice to write the formula once as an equation than to write multiple *to-decide* functions, one per variable.

Units have difficulty with aggregates such coordinates. For example, the following formula is defined using all numbers, because even though the types of *x* and *y* are different, and *y* and *b* are the same, Inform's dimensional analysis will not allow them to be combined.

```
Equation - Slope-Intercept Formula
        y = mx + b
where y is a number, x is a number, m is a number, and b is a number.
```

Backstage Activities

Actions are player-generated events. Activities are library-generated events. Typically they print something that an author might want to customize for their work. Parsing, prose generation, complex actions, and a few miscellaneous events to bookend the game comprise the bulk of activities. It's interesting that what events a library exposes is indicative of what the tool considers, and has considered, important over the years.

Each activity is composed of three rulebooks, whose names begin with *before*, *for*, and *after*. The default outcome of the *for* rulebooks is success, so only the most specific applicable rule will fire. The other two run all applicable rules within themselves. We can test to see if an activity in progress via *while* in the rule preamble. This allows us to vary narration of the same object or event differently depending on context.

The optional word *rule* may begin a *for* rule as syntactic sugar, and the usual *of, for, rule* may be inserted before the parameter. For activities with have articles in their name, such as *printing the name*, those articles are required. An activity's name allows no leeway in its wording.

Explanations of each activity, and its general category, follow.

Prose Generation. This may be a heavy label for what is essentially just outputting the contents of text properties, but they are some of the most important library events in Inform. The current contents of these rulebooks in a given work are listed in the "How Things are Described" section within the Rules tab of the Index.

Printing the name of something. This prints the printed name property. As many times per turn as this activity is called, it's more efficient to put say-phrases in the *printed name* property than create a rule that runs when every other object's name needs be printed. Still, to change the name for a set-description, or for unusually complicated situations, it is still one of the most used activities in Inform.

> Rule **for printing the name** of Joan when Joan has not managed the player: say "important-looking woman".

Printing the plural name of something. This prints the *printed plural name* property. This is only used from within the following *printing a number* activity, right after said number is printed.

Printing a number of something. Used for indistinguishable instances of a class, because a list of "a coin, a coin, a coin" is pretty dull, and normal instances each have a distinct name of their own. Skipped if there's only one such in the location. The variable *listing group size* holds the number, and is useful with the say-phrase in words. Calls the above *printing the plural name* activity.

Listing contents of something. This very useful library event pretty-prints a list of objects contained within the sole object passed to it, such as the player or the location. It's used by

taking inventory and for the you-can-also-see sentence of *looking*. Its before rules provide the phrases *group (set-description) together*, *group (set-description) together as (text)*, and *group (set-description) together giving articles*, which ensures a multitude of instances appear next to one another in the list by calling the following activity grouping together.

Grouping together something. Utilized by the above to wrap or replace the members of the group with other text. A *for* rule here will override the text used in *group (description) together as (text)*, and is a good way to treat multiple distinct instances as if they were indistinguishable instances (such as our coins example). The variable *listing group size* again holds the number of group members. The passed-in parameter is only the first member of the group, and is otherwise insignificant.

(The multiple activities belonging to *looking* are in their own section further down.)

Parsing Input. Doing a good job with these activities can make or break a game. The contents of all these rulebooks are listed in the "How Commands are Understood" section of the Rules tab of the Index.

Printing a parser error. A distressingly important activity, it allows narration of the nineteen different parser errors Inform can throw, none of which are terribly informative for how the player should re-factor his input. While extensions such as Default Messages and Custom Library Messages can change the parser errors themselves, typically it is useful to vary the errors based on situation and available information, to give the player much better, much more specific feedback.

> **For printing a parser error** when the latest parser error is *I beg your pardon*: try looking.

Reading a command. Each of the three phases of this activity is useful. *Before* is useful for initializing variables for the next turn. *For* rules can actually skip reading the player's command, allowing us to insert text into the command buffer directly and use that instead, via the special-case phrase *change the text of the player's command to (indexed text)*. *After* rules are useful for massaging the input before it is fed to the parser. Note that although Inform supports regexes, they are slow, so use the following standard phrases if at all possible. The forward slash denotes high-precedence altercation.

> **After reading a command** when the player's command includes "please/thanks", cut the matched text.
> **After reading a command** when the player's command matches "take a nap", replace the matched text with "nap".
> **For reading a command** when the command to retry is not "":
> **change the text of the player's command to** the command to retry;
> now the command to retry is "".

Clarifying the parser's choice of something. When the parser takes a guess at what the player meant (via the *does the player mean* rulebook), it says so in a parenthetical like, "(your coat)". We can change this message, or for bleedingly obvious cases, censor it. The variable *the item described* refers to the parameter.

For clarifying the parser's choice of your coat: do nothing.

Asking which do you mean. When the parser cannot guess which object was meant, usually because the name the player used applies equally to either one and *does the player mean* was no help, it ask, "Which X did you mean?" Though writing a *for* rule would be a major undertaking, it's use may be in *printing the name* as a condition, *while asking which do you mean*.

Supplying a missing noun. Sometimes a verb is used without a noun because the noun is obvious in some way. For example, EAT when there's only one edible thing in the location. By informing the grammar rules that a verb can be entered without the noun, this activity will be called to assign something to *the noun*. If *the noun* is still nothing after the activity finishes, then the normal parser error will result: "You must supply a noun."

> Understand "eat" as eating. [as opposed to "eat [something]"]
> Rule **for supplying a missing noun** while eating: now the noun is a random edible thing in the location.

Supplying a missing second noun. As above.

Deciding whether all includes. When the player enter TAKE ALL, it really shouldn't be every object. The drapes, the sun, the kitchen sink, Bob – these should be excluded. By default, other people, *scenery*, and *fixed in place* (i.e., nailed down) objects are immune to ALL, but sometimes we wish to disallow other things. The imperatives *it does* and *it does not* may be used instead of the usual.

> Rule **for deciding whether all includes** something which frightens the player: it does not.

Deciding the scope of something. Used with the phrases *place (object) in scope; place (object) in scope, but not its contents*; and *place the contents of (object) in scope.* When the visibility and accessibility rules miss an object, this hook is something of a hack. Generally, only the *after* rules are used.

Deciding the concealed possessions of something. Normally, the parts and carried contents of people, machines, etc. are plainly visible. But sometimes we wish the *particular possession* of the person (or whatever) to be invisible.

For deciding the concealed possessions of the secret assassin when the particular possession is the dagger: yes.

Taking. This action is the most common action to cause a change in world-state. In the text adventure days, it was fashionable to simulate the physical world to such a degree that one had to open doors before going through them, unlock doors before opening them, and *taking* a nearby item before using it. As time went on, this level of fine detail lost its novelty value, and nowadays most players are annoyed if their character doesn't have the sense to automatically open a door before walking through it. Automatically taking a nearby object

before eating or otherwise using it has been bugging players long enough that the library provides for it.

Implicitly taking something. If the *for* rule fails, then TAKE won't automatically be done. This rulebook is listed in the Index's Rules tab, at the bottom of "How Commands are Understood."

System level. These events bookend the game and are rarely needed. But they occur in places that user-created code would be hard-pressed to get into had the library not specifically exposed them, so it's just as well that they're offered.

Constructing the status line. We can already change its contents by assigning to *the left hand status line* and *the right hand status line*, but it's sometimes useful to use shorter names up there due to space constraints.

```
For printing the name of the fourth diagonal corridor of the sixth floor while
constructing the status line:
    say "6th floor"
```

Printing the banner text. This prints the title, subtitle, author, and version information at the opening of play. Messing with this is a quick route to making a game look amateurish. But sometimes an *after* rule is tasteful.

Printing the player's obituary. Prints the "you have died" message followed by the score, if any.

Amusing a victorious player. Prints text in response to the command AMUSING at the game over prompt.

Starting the virtual machine. Multimedia or specific VM interpreters may need this early event, but that's about it.

Looking. This slightly-recursive action does the most prose generation of anything in Inform. In text adventures of old, *looking* was your primary vehicle for delivering important information. Nowadays *scenes* can carry much of that weight, but the library still dedicates much of itself to ensuring which closed containers are lit and which aren't, and what you can see from multiple vantage points. Each level of recursion is called a *locale* and has two parts: items with an *initial appearance*, each getting a paragraph to itself, and the nondescript items, which are in a single paragraph together. When we stand on a stool inside a bear pit inside the room, *looking* will articulate what's what at each level of recursion, starting with the outermost and working its way inward. (A work's current rules for these activities are listed in the "How Things are Described" section of the Rules tab of the Index.)

Looking uses two properties for bookkeeping: *mentioned* and *marked for listing*. Things are initialized to *not mentioned* early in the process, and after printing an item's *initial appearance,* or if the item was used in *writing a paragraph about*, the item is then *mentioned*. After the *interesting locale paragraphs rule* finishes iterating, the leftover items still *not*

mentioned are initialized to *marked for listing*, so the final *listing nondescript items* activity can list them in a single paragraph.

Initial appearance, by the way, is set by the text that floats by itself after an item.

> The worn-out tennis shoes are a wearable thing. **"Your trusty tennies lounge by the front door."**

The behavior is deep enough to require an illustrated callgraph of sorts. A rule doesn't have the --> in front of it. Activity names are in bold. Parenthetical remarks provide information about the rule's purpose or activity's parameters.

[the *carry out looking* rulebook]
-- room description heading rule (prints the room's *printed name* property)
-- room description body text rule (prints the room's *description* property)
-- room description paragraphs about objects rule (it doesn't really recurse, but iterates on ever-smaller domains)
-----> [**BEFORE Printing the Locale Description** of *the room, container, or supporter*]
-----> -- initialise locale description rule (initializes objects to *not mentioned)*
-----> -- find notable locale objects rule
-----> -----> [**Choosing Notable Locale Objects** for *the room, container, or supporter*]
-----> -----> -- standard notable locale objects rule (uses *set locale priority of (object) to 5* on each object)
-----> [**FOR Printing the Locale Description** of *the room, container, or supporter*]
-----> -- interesting locale paragraphs rule (sorts objects in *locale priority* order)
-----> -----> [**Printing a Locale Paragraph about** *the prop*]
-----> -----> -- don't mention player's supporter in room descriptions rule (marks it *mentioned*)
-----> -----> -- don't mention scenery in room descriptions rule (marks it *mentioned*)
-----> -----> -- don't mention undescribed items in room descriptions rule (marks it *mentioned*)
-----> -----> -- set pronouns from items in room descriptions rule
-----> -----> -- offer items to writing a paragraph about rule
-----> -----> -----> [**Writing a Paragraph About** *the prop*] (empty; for author use)
-----> -----> -- use initial appearance in room descriptions rule (prints the *initial appearance* property)
-----> -----> -- describe what's on scenery supporters in room descriptions rule ("On the scenery-supporter is...")
-----> -- you-can-also-see rule (initializes *marked for listing* and calls activity)
-----> -----> [**Listing Nondescript Items** for *the prop*] (says "[In/On the supporter/container] you can [also] see...")
-----> -----> -----> [**Printing Room Description Details** of *the prop*] (but *the item described* is the room)
-----> [**AFTER Printing the Locale Description** of *the room, container, or supporter*] (empty)
-- check new arrival rule

Printing the locale description of something. This is the big, overarching activity within the *carry out looking* rule *room description paragraphs about objects*. If we stand on a stool, in a bear pit, in the room, this iterates through those "locales", from outside to inside, so it feels like recursion without actually being so.

Choosing notable locale objects for something. This initializes each item's local priority to five. In an *after* rule, we can *set the locale priority of (object) to (number)* to enforce a particular ordering of the objects. One is high priority. The activity is called once per locale – the parameter is the room (or whatever) – so a *for* rule on a room should set priorities on all objects in it. A priority of zero is not mentioned.

Printing a locale paragraph about. Finally, it's time to start printing prose to the screen. Well, not all objects deserve any mention whatsoever. The player is one of them – it is *undescribed*. This activity will 1) filter out objects with the *scenery* and/or *undescribed* properties, then 2) filter out the supporter the player is on (since it was mentioned in the room heading), then 3) offers the remaining items to *writing a paragraph about*. If that activity had nothing to say, then it continues by 4) printing the *initial appearance* property, if any, and then 5) if the object is a scenery supporter with non-scenery stuff on it, it will describe the supporter and its contents.

Writing a paragraph about. Initially empty, this is intended for the author's use. It overrides *initial appearance.* As is usual, say-phrases in properties execute faster than rules which must be considered. This activity is called several times per LOOK.

Listing nondescript items of something. Items lacking *initial appearance* or a *writing a paragraph about* rule will be *marked for listing*, then fall into here. This activity prints one sentence, via the *listing contents* activity. This activity's line may begin with "On the (supporter), you" or "In the (container), you" if the locale isn't a room.

Printing room description details of something. Some items' names are trailed by a parenthetical filled only with properties like *open, closed,* or *empty*. This only occurs in the you-can-also-see line of a room description. This activity allows changing these parenthetical messages. (With the Default Messages extension, these are library messages 101 through 107.)

Darkness. Like bottomless pits in platformers, darkness has been one of the commonest tropes in text adventures, stretching back to its earliest caving days. While in 2010 darkness can be simulated with a single *instead* rule, the library has always supported it, and supports it still. The following five activities each only print a message, any of which could be replaced with extensions like Default Messages or Custom Library Messages. While these activities might be useful if we create a game involving heavy use of light and dark – replacing the default message with a say phrase that calls a rulebook would give the same effect – we can otherwise ignore their existence. They take no parameters. The truth state variable *the darkness witnessed* remembers if the player has ever been in the dark.

Printing the announcement of darkness. "It is now pitch dark in here!"

Printing the name of a dark room. "Darkness"

Printing the description of a dark room. "It is pitch dark, and you can't see a thing."

Printing a refusal to act in the dark. "It is pitch dark, and you can't see a thing." The visibility rulebook decides if this activity is called.

Printing the announcement of light. Performs try looking.

New Activities. Occasionally we wish to create an activity of our own. Extensions tend to find this useful more often than game-specific code, but if some kind of complex interaction is complicated enough, or is worthy of altering narration via *while*, it may warrant an activity. An activity takes at most one parameter, of any type. *Something* must exist if it takes a parameter, and *on type* must exist if the parameter type is not *object*.

Planning *something* **is an activity** *on* people.

Drama management *something* **is an activity** *on* scenes.

The name of the activity will always have *before, after,* or *for* in front of it, so name accordingly. The *for* stage of an activity should, by default, have a single *last* rule which provides the default behavior. Other *for* rules will always fall in front of the default, getting first shot at handling the situation. Again, the words *of, for,* and *rule* may appear or not between rulebook name and parameter description.

Last **for planning** for someone (called our thinker): say "[Our thinker] can't think of anything to do." instead.

Finally, the phrase *carry out the (activity) with (parameter)* invokes the activity. It must appear somewhere in our code or our activity will never be used. The *the* is required.

Before drama management for the entire game, **carry out the** *planning activity* **with** *the player*.

Testing Commands

We have one testing command in the language itself:

```
Test me with " look / test foobar / wave ".
Test foobar with " look / take me / jump ".
```

This will execute the slash-separated list of runtime commands when the TEST runtime command is issued. *Me* is the usual name for the top-level test script.

We have several run-time testing commands at our disposal.

SHOWME an object: Will list the current values of all the properties of that object, including any relations it's involved in.
RULES: Will list the name of a rule before it executes.
RULES ALL: Will list the name of a rule before it considers execution.
RULES OFF: Turns off the mode.
ACTIONS / ACTIONS OFF: Will list when actions begin and end.
RELATIONS: Lists the current state of all relations.
SCENES / SCENES OFF: Lists the scenes currently happening, and will update us when one begins or end.
TREE: Shows all instantiations, indented according to the containment relation.
TEST a test script: Runs the list of commands.
PURLOIN an object: Immediately places the object in the player's inventory, no questions asked.
ABSTRACT object TO object: Similar to purloin, but gives the first object to the second object, which is likely to be a container or some such.
GONEAR an object: Moves the player to the room which has the object.
RANDOM: Random number generator now predictable.
SHOWVERB: Shows I6-level information about a verb: synonyms, arguments, whether the arguments are reversed for a particular line.
SCOPE: Show all objects currently in scope.
SCOPE object: Show the scope from the particular object.
TRACE a number: Shows very low-level parsing information. The number ranges from 1 (the default) to 6 (the most detailed).

The IDE has the *skein* -- which is a tree of commands of every playthrough we've done in the IDE -- and the *transcript* -- which is a detail view of one path in the skein. It has the output text from those commands. The transcript also has the *bless* button, so automated re-testing of changes is possible.

Times, Turns, and Tenses

Narratively, flags and counters are some of the most useful pure-coding constructs in interactive fiction. Varying the prose on what has or hasn't been done before, or at all, has been so important to the form (and cluttering source code for so long) that Inform counts and flags many things automatically, providing the results to us in the form of specialized syntax. Besides 1) the global *time of day* and *turn count* variables, Inform automatically tracks 2) how long until a scheduled event will occur, with *at* rules, 3) how many times an action was attempted, or a condition became true, with *for the Nth time*, 4) for how many turns in a row a condition stayed true or an action was attempted, with *for N turns,* 5) if an action has ever succeeded on an object or a condition has ever become true, via the perfect tenses, and 6) the world-state as it was on the previous turn, via the past tenses. Once learned, these become major tools in the construction of a quality game and very readable source.

1. The variables *the time of day* and *the turn count* hold the current value for each. We test these same as any variable.

> Instead of going when **the time of day** is before 5 pm, say “It ain't quittin' time yet.”
> After looking when **the turn count** is one, say “And so your adventure begins.”

2. We can schedule an event (a rule) to fire in the future. Method one is hard-coding an in-game time. It fires only once.

> **At 12:00 PM:** say "The lunch bell rings!"

Method two is more flexible because it can be re-scheduled. First, we name the event. In the following example, *the watch will beep* is a rule, which lives in an opaque, unnamed pseudo-rulebook only while it is scheduled. (Else, it's unlisted.)

> **At the time when** the watch will beep**:** say "Your Casio beeps at you urgently."

Then we schedule it imperatively. The *at* phrase takes type *time*, while *in* takes a number of either turns or minutes. Note that *turn*/*turns* is part of the phrase, while *minutes* is part of the *time* type. A turn is an action, and is related to time by the ratio of one turn to one minute, though extensions such as Phrases For Adaptive Pacing can modify this.

Carry out waiting until:	the watch will beep **at** the time understood.
Carry out waiting for:	the watch will beep **in** the time understood **from now**.
Carry out pushing the watch:	the watch will beep **in** six minutes **from now**.
Carry out pushing the watch:	the watch will beep **in** six **turns from now**.

An event can safely re-schedule itself.

> When play begins, the alarm clock goes off **at** 6:30 am.
>
> **At the time when** the alarm clock goes off**:**

say "Your alarm clock wakes you up.";
the alarm clock goes off **at** 6:30 am.

3. Inform counts how many times an action was attempted which our rule preambles query like the following.

Before taking the candle **once**: say "Hey, it's a candle."
Before taking the candle **twice**: say "You guess you'll need that candle again after all."
Before taking the candle **for the third time**: say "Your life seems tied to this candle."
Before taking the candle **for at least 4 times**: say "Sigh."

These action amendments count both successful and unsuccessful attempts, so we should confine them to rules guaranteed for consideration every turn: *before* and *every turn*, and early in *persuasion* and *instead*. Any later than that and we must take precaution that the rule isn't pre-empted, missing the chance to even see the Nth attempt.

Check burning when the candle is unlit **once**: say “You'll need to light the candle first.” instead.
Check burning when the candle is unlit **twice**: say “Again, your source of fire needs be lit first.” instead.
Check burning when the candle is unlit **for at least the third time**: say “The candle is out again.” instead.

After burning the candle **twice**: say "If a *check* rule blocks the second attempt, this *after* rule never fires. But even if this rule does fire, it isn't necessarily narrating the second time the candle was successfully lit up. The first attempt may have failed, so this rule would be firing on the first successful attempt."

The *for the Nth time* amendment can also be added to relations and conditions to check how many times they become true.

Every turn when the candle is lit **for at least the third time**: say "The oft-used candlewick lights easily."
Every turn when the player carries all the gear **for the third time**, say “How did I get stuck being the pack mule?”

4. Inform also counts for how many *turns [in a row]* something has remained true. The syntax uses the word *turns* rather that *times* for this, as it's a measure of duration rather than a discrete tally of occurrences. If used on actions, it means that the action was done multiple times *in a row*. Indeed, to keep the distinction clear, it's helpful to mentally amend the words *in a row* to any *for X turns* amendment.

Every turn when the player does not carry the sprite **for at least 3 turns**: say "The sprite sings a lonely song."
Every turn when the candle is lit **for five turns**, say “Your candle burns on.”
Every turn when the player carries all the gear **for ten turns**, say “Isn't it someone else's turn to carry this stuff?”
Check jumping **for three turns**, say “You ain't gettin' any higher.” instead.

It's rare to see a *turns* amendment directly on an action as we've done with *jumping*, because it's rare for a player to type in the same command three times in a row and expect different results. *For X turns* make more sense with relations, properties, and conditions because they have a tendency to hold their value over very long periods of time. By contrast, *for X times* makes sense on nearly anything, though when used on actions needs to occur early in processing so *check* rules and whatnot don't rob it of its only chance to fire.

The narrative use of these amendments is for varying the narration of repetition. Interactive fiction differs from video games in that a first success is worth reading about, and the second only to ensure that you've learned the skill, but after that why tell the reader again. It's old news. Of course, repetitive failure is not much fun either, but that at least has a narrative role as struggle. And perhaps a gameplay role too, as puzzle-solving.

5. While minutes and turns both precisely measure time, tense is a relative measure expressed in English via verb inflections.
A few of these inflections Inform gives a meaning to. The perfect tenses mean "have ever been true". So an action in the present-perfect, *if we have taken the candle*, begins the game false, becomes true on the first successful TAKE, and never again changes value. Relations, like *if Bob has carried the candle* work similarly, but the implementation differs. Each object has a hidden bit array property holding the answers to *if we have* queries, so questions of the form *if we have taken the noun* work fine. Relations are case-by-case: an implicitly-created variable `BobHasCarriedCandle` holds this answer. So actions accept variables like *the noun* but no subject – the *we are* is literal, required, and has no synonyms – while relations accept a subject like *Bob* but no variables – `BobHasCarriedNoun` just won't work right.

> if Grognar **has carried** the Sword of Smiting,
> if Bob **has ever remembered** the time he spent in jail, [present is *can remember* – *can* becomes *ever*]

The perfect tenses, like stories in general, are about irrevocable developments. While the condition *if Grognar carries the Sword of Smiting* is correct when we're checking to see if he can swing the sword, it isn't correct if an character asks him to describe the sword. Sure, if he has never encountered the mythical blade he has no first-hand knowledge of it, and if he does carry it then he certainly can describe it. But if the sword is temporarily separated from him, asking him under the simple present tense produces a false negative. Grognar has learned what the blade looks like, and that is as irrevocable as discovering the villain is his own father. This then is where the perfect tenses come in: knowledge and plot points.

Before moving on, a quick detail about testing the present for the sake of syntax consistency: when testing whether or not a particular action is currently going on that turn, *we are* is accepted in front of the action, but is optional. This mirrors the *we have* of the present-perfect.

> Every turn: if **taking** something unique, [perfectly synonymous]
> Every turn: if **we are taking** something unique,

6. Relations also support the past and past-perfect tenses. If you've ever implemented a linked list, you likely know how useful a "previous" node pointer is in addition to the "current" node pointer, especially for updating those very node linkages. Similarly, when the condition of an

if-statement is asked in simple past tense, it queries the previous turn's game state, not the current one. For example, when phrased in present tense the following lines of code would be broken: the candle would always be lit afterward regardless its state beforehand. But written in past tense, the lines indeed toggle the *lit* property.

```
if the candle was lit, now the candle is unlit;
if the candle was unlit, now the candle is lit;
```

(Properties use the verb relation *is/was*. Again, properties are a special case of relations.) Remember that by itself the simple past tense only asks a question of the immediately previous state, not necessarily what changed since the previous turn. So the following wouldn't do what was intended.

```
if Bob managed the player, say "So long, Mr. Picky!"
```

When Bob was the manager on the previous turn, most likely it is because he is the manager now, too. Try this instead.

```
if Bob managed the player and Bob does not manage the player, say "So long, Mr.
Picky!"
```

We recognize change by bracketing it with two world-states, the present one and the previous turn's (past) one. And should the player again come under the tyranny of Bob, the message will be re-printed when he yet again leaves Bob. Now we can greet our new manager.

```
if Joan did not manage the player and Joan manages the player, say "Hello, Joan."
```

The past-perfect (*if X had been Y* or *if Bob had managed Dave*) relates to the present-perfect as the simple past relates to the simple present: the past-perfect remembers the previous turn's present-perfect value. Because the present becomes the past at the end of a turn, during the *update chronological records rule* found in the *turn sequence* rulebook, and because the perfect tenses do not change once set, the only time throughout the entire game when the past-perfect isn't equal to its present-perfect twin is during the end of the one turn that changes the present-perfect.

To continue Joan's example, replacing the past and present with the past-perfect and present-perfect still recognizes the change via bracketing, except this time the rule will fire at most once throughout the entire game.

```
Every turn, if Joan had not managed the player and Joan has managed the player,
say "Pleased to meet you."
```

The condition must appear in *every turn* or as a scene begin/end point because that is the only time during a turn that past-perfect differs from its present-perfect twin.

Edge cases. For syntactical sugar, these simple things sure have a lot of edge cases. It isn't so much a case of bad programming as realizing just how much common sense and filtering happen when people speak naturally.

Negating the Perfect Tenses. We cannot do the following.

> Every turn, if Bob **had managed** the player and Bob **has not managed** the player, say "So long forever!"

It's nearly identical to the above example with Joan, except the *not* moved. Now that rule never fires. For that rule to fire, the perfect tense would need to transition from false to true, which perfect tenses don't do, by definition. *Had* is always what *has* used to be. So in the first condition *had* is saying that *has* used to be true, while the second condition says that *has* is false right now. It's a contradiction.

Even when used alone, negating the perfect tenses can be counterintuitive. Constructions like *if we have not trusted* do not mean "if there was a time when we did not trust", but rather, "if we have yet to trust". It is true at the start of the game and becomes permanently false the first time we trust, as opposed to being false at the start of the game and becoming true the first time *distrusting* happens. (For that effect, a *distrusting* action really is needed, and likely a relation the twin actions set and unset so it is possible to tell which action happened the most recently.)

If this isn't clear, I can only recommend circuit timing diagrams. Suffice to say, the perfect tenses are good for knowledge and plot points, the past tenses are good for recognizing world-state changes by bracketing, and when put together we can recognize additions to knowledge.

Combining tenses with occurrences. By these rules, it seems rules like *if we have taken the candle twice* would never fire, because *we have taken*, once becoming true, stays true for the rest of the game. Since it doesn't become false again, it cannot become true again, so *twice* is never satisfied. Fortunately this is special-cased to work as it looks like it should.

> Every turn when **we have eaten** the slow-acting poison **for 5 turns**: say "You start to feel ill."
> Every turn when **we have eaten** the slow-acting poison **for at least 6 turns**: say "You still feel ill."
>
> Check listening to the ominous skritching sound when **we have taken the candle at least three times**:
> say "Perhaps your life really is tied to this dim, flickering candle." instead

Initial State. The *update chronological records* rule is ran just before the *when play begins* rules to set up the initial state. Prior to this very early rule, everything is still zero, or possibly garbage, and the player is in the dark room, "thedark", so testing the past via any of this chapter's features may not be a good idea. It is possible to explicitly *follow* the rule again, or to list it after the *position player in the model world* rule to correct snafus, but this has other effects. For example, throwing off the count in things like *for two turns* when connected to the beginning of play.

Timeless Actions. *Out of world* actions do not affect time, but Inform still counts their occurrences. So we could do this.

> Check saving the game **once**: say "You have two more chances to save the game."

Check saving the game **twice**: say “You have one more chance to save the game.”
Check saving the game **for at least four times**: say “Sorry, you've already used up all your saves.” instead.
Check quitting the game when **we have not requested the score**: say "Not curious about your score?" instead.

But we couldn't do this.

Check saving the game **for at least two turns**: say “Saving the game multiple times in a row is a bad idea, because you only have a limited number of saves.” instead.
Check quitting the game **for two turns**: say “All right, all right, you don't want to play anymore. Fine.”

This is because if we clue the player that doing something thrice in a row will have negative consequences, not only will they be certain to try it, but they'll type SAVE just before the third and final attempt.

Verbs Can Pull Double-duty. It's possible to define the same verb as both action and relation. The closest to ambiguity that can be reached is the ungrammatical *if Bob watching Junior*, which means the same as *if Bob is watching Junior*, which is clearly a relation. Without the relation, the former will not even compile. NPC actions in *if* statements require *trying*: *if Bob trying watching Junior*. Likewise changing the subject to *the player*, which will always be a relation test, or *we are/have*, which will always be an action test. Since the purpose of almost all actions is to set or unset a relation, using the same verb for both the relation and the action which sets it reduces our cognitive load.

Sometimes the word *trying* needs to be inserted even when there is no possibility of ambiguity, such as in a table column of stored actions.

Past Participial Adjectives versus the Passive Voice. In English, the passive voice swaps the subject and direct object of the sentence. The preposition *by* usually precedes the former subject, assuming it isn't dropped entirely.

Clarissa drew the portrait.
The portrait was drawn by Clarissa.
The portrait was drawn.

The verb phrase changes to the *be -en* form: *is drawn, was drawn, am drawn, are taken, were driven,* etc. But most English verbs don't have an -en (past participial) form, so they use the -ed (past) form in its place: *is painted, was closed, am carried.* It is also true that the past participle -en form can be used as a plain adjective: *the drawn portrait, the taken picture, the closed book.* These rules collude to produce a grammatical ambiguity in English.

The door was closed.

That sentence might be in passive voice with a dropped subject, as in *the door was closed by someone*, or might be an ordinary assertion in simple past tense, as in *the door was not ajar.*

Inform supports the passive voice only in one instance: relation verbs declared as *to be able*

to, such as our *can remember* example. Passive voice can be stated in the simple present or past tenses: *the time he spent in jail can be remembered by Bob* and *...could be remembered by*. Inform's relations do not support passive voice in the perfect tense, nor passive voice actions, nor any other passive voice combination. Crucially, this one instance of passive voice that Inform does implement still requires both parameters. We cannot say "if the time he spent in jail can be remembered". We need the *by someone*.

However, Inform does support adjectives, and adjectives can be named for a past participle. *Carried* is one such example. Though the standard rule define *carry* as a relation, it also defines *carried* as an adjective via *Definition*. So both of the following are valid lines of Inform code, but for entirely different reasons with entirely different meanings.

```
    if the candle is carried,                [ An adjective test, not a relation test in passive voice. The player is implied. ]
    if Clarissa carried a candle,    [ A relation test (tipoff: two parameters) in simple past tense. ]
```

An object is *carried* only if the player carries the object. "A person *carried* an object" only if that person was carrying the object on the previous game turn.

Chapter 9.13 of the manual, Writing With Inform, gives this shining example for the simple past tense.

```
    if the lantern was switched on, now the lantern is switched off;
    if the lantern was switched off, now the lantern is switched on;
```

Like *carried*'s dual definitions, *switched on* has dual definitions: a *device* object has an adjective property *switched on/switched off*, and the present-perfect tense of the *switching on* and *switching off* actions is *we have switched on* and *we have switched off*. But even if we also created a *to be able to switch on* relation in addition to the property and the action, there will still be no ambiguity in Inform. *We have* always identifies present-perfect actions, two parameters surrounding a verb identifies relations (even for passive) except for when *trying* separates subject and verb (which is an NPC present action), and one parameter identifies the adjective. A word ending in *-ed* does not guarantee past tense – it could be the present-perfect. And verb constructions of the form *be -en* don't always identify passive voice – it could be a past participial adjective in simple past.

Swapping Headings

As part and parcel of literate programming, Inform provides headings for source code. One of *volume, book, part, chapter,* or *section* against the left margin denote the line as a heading.

> **Volume 1 -** Technical Tidbits
>
> **Book 1 -** Definitely Necessary Definitions

The IDE puts the heading in bold type, and the heading appears in the contents tab above the source's pane. (Not to be confused with the contents tab in the index.) The contents shows our source as an outline, and can even narrow the view to just a particular portion of the whole code. When used this way, headings are merely a way of organizing our code, and the index.

Yes, the index, the usefulness of which is easy to overlook. For those of us uncomfortable with *rules*, it shows in what order the rules are sorted. For those of us uncomfortable with abbreviating object names or putting spaces in identifiers, it shows all types of all constructs. And it shows what constructs we created by typo. Even though variables are grouped with other variables and phrases with other phrases, each construct, in the index, is grouped under the most specific heading it follows.

For the extension writer, the index is highly important because the users of your extension may read the documentation only once, and thereafter read the pithier index to refresh themselves on the extension's abilities. Careful choice and naming of headings, and careful arranging of source, create index listings that serve as flash cards. Combined with sensible synonyms on phrases and functions, an extension's functionality becomes easy to remember, easy to guess at, and quick to look up.

Inform provides a few features that work on headings and their contents. One is *unindexed.* Simply put, most constructs within an *unindexed* heading don't appear in the index. Rules will always appear, for instance, but scratchpad variables and utility functions can be hidden. While Inform has no namespaces to speak of, merely not advertising the existence of such works well enough, and the extension's complete source is easily seen anyway. Again, Inform is a white-box language.

> Section 8 - odds & ends - **unindexed**

Second is *not for release*. Testing and debugging commands are a fact of life, but should a player discover one it could harm the game or story. So this heading amendment removes everything in its section: rules, objects, everything. Unlike *unindexed*, subheadings also fall under a parent heading's *not for release*.

> Chapter 6 - testing commands - **not for release**
>
> [any *section*s within this chapter also remain unreleased]

Headings can also remove, add, or replace source text with the source text of other headings via the parenthetical amendments *for use with, for use without,* and *in place of*. This is typically done to modify the extensions of others without editing the files, greatly simplifying administrative tasks such as versioning or "keeping around an extra copy". These, too, affect their subheadings. There is a slight restriction on *in place of*: when replacing a heading, the replacement must be the same type of heading (*chapter, section,* whichever).

> Chapter 2a - automatic safe cracking **(for use with** Locksmith **by** Emily Short**)**
>
> Chapter 2b - mostly automatic safe cracking **(for use without** Locksmith **by** Emily Short**)**
>
> Section 6 - hacked locking **(in place of** Section 1 - Regular locking **in** Locksmith **by** Emily Short**)**
>
> [this is a *section* because it is a *section* being replaced]

A good extension writer will break up an extension into named sections to facilitate heading replacements. (This is of course somewhat at odds with beautifying the index.) One extension is automatically included in all projects: the Standard Rules by Graham Nelson. It contains the most basic information on the class hierarchy, the built-in actions, variables, etc., of Inform 7. For example, the properties on class Thing are here.

> Section SR1/3 - Things
>
> A thing can be lit or unlit. A thing is usually unlit.
> A thing can be edible or inedible. A thing is usually inedible.
> A thing can be fixed in place or portable. A thing is usually portable.
> A thing can be scenery.
> A thing can be wearable.
> A thing can be pushable between rooms.

And so on. Though it will likely cause compiler errors in many places, the standard rules can be modified like any other.

> Section 8 - my thing class (in place of Section SR1/3 - Things in **the Standard Rules** by **Graham Nelson**)

Avoiding compiler errors by changing so much of the most basic assumptions of the code library is difficult, but we're hardly new to compiler errors.

Facing Inform 6

All of an Inform 7 project compiles to Inform 6 source code before compiling again to assembly. This means all of Inform 7's constructs are implemented in Inform 6 somehow. For example, a rule is a boolean function that takes no passed-in parameters -- action variables like "the noun" and "the second noun" are global, while local variables are on the stack -- so a rulebook is an array of function pointers. Larger structures such as actions and activities are composed of several rulebooks. Object instantiations are connected together by linked-list pointers, so a set-description may loop through them. Each set-description compiles to a unique function containing a loop header, and like generator functions in functional programming, they return the first applicable value that passes muster. (The calling code, which has the loop body, is responsible for informing the set-description where it left off so it can find the next applicable value.) Meanwhile, scenes, which can have properties just like objects, are implemented as several different arrays, one per property. So a scene name is just a named value, an index into those arrays. And because a single table column's entries all have the same type, a table column is an array plus a few header bytes, while a table is an array pointing to those columns. And so on and so forth.

The implementation for all these constructs is found in the template files within the Inform application. These text files, which have the extension *i6t*, we can modify from our Inform 7 source similar to section headings. We can replace parts of them, or insert additional bits between them.

```
Include (- ... blah blah blah... -) before "Relations.i6t".
Include (- ... blah blah blah... -) instead of "Relations.i6t".
Include (- ... blah blah blah... -) after "Symmetric One To One Relations" in
"Relations.i6t".
```

Usually we just need to pull out a bit of useful information or add in a useful line in the generated source. The commonest way is by creating a to-phrase or to-decide function whose entire body is Inform 6 code between the (- and -) markers. This inserts the exact text into the compiled Inform 6. So, generally, to-phrases must end in a semicolon, while to-decide functions, which are frequently if-conditions or r-values, must not.

```
To decide what number is the chosen table row: (- ct_1 -).
To decide what number is the first misunderstood parser word: (- (wn - 1) -)
To decide which number is (x - a number) ORed with (y - a number): (- ({x} | {y})
    -).
To really clear the screen: (- VM_ClearScreen(0); statuswin_cursize = 0; -)
```

Parameters go between curly braces. There's two parameter types useable here unavailable to pure Inform 7. *Condition* is something appropriate to follow an *if*. *Action* compiles to an invocation of *TryAction()*. Inline assembly may be included amongst the Inform 6 like any other statement. Assembly opcodes always begin with a @ sign.

```
To do this: (- do { -).              [ not ending with a semicolon here, obviously ]
To until (C - a condition): (- } until {C}; -).
```

To push (x - a word value) onto the stack: (- @push {x}; -).
To pull (x - a word value) from the stack: (- @pull {x}; -).

Other Inform7 variables may be accessed by placing the name between (+ and +) markers. Rules may be invoked by following the markers with empty parenthesis, since rules never take a parameter. Named to-phrases expand, not to the function, but to an array of metadata about the function. Index 1 holds the function pointer, so the syntax would resemble ((+ my I7 phrase +)-->1)(... any parameters ...) while indexes 0 and 2 hold the type and printable name, respectively.

An extended example: the author wants to save actions for later display or execution from a hint system or CYOA menu. One goal is as always a nice Inform 7 syntax, preferably without using *the action of* keywords that herald a stored action. So he reaches for the *action* parameter type, which requires an Inform 6 inclusion, then reflects the Inform 6 right back into an Inform 7 to-phrase, which has all five parameters of the *TryAction()* invocation, plus two new parameters: a description of people, and whichever rule is currently held in his *extra behavior* variable.

The first line here shows the resulting syntax within a Before rule.

```
Before an actor opening something locked, a strong person in the location could try attacking the noun.

The extra behavior is a rule that varies.

To (P - a description of people) could try (invocation - action):
    (-  Could{invocation}{-backspace}{-backspace}, {P}, (+ extra behavior +) );
    -).

Include (-
[ CouldTryAction req actr act n sn desc exbehvr;
    ((+ remembering for later +)-->1)(req, actr, act, n, sn, desc, exbehvr);
];
-).

To (requested - truth state) intention by (dont-use-this - a person) considering (possible-action - an action name) via (possible-noun - an object) & (possible-second-noun - an object) by any able-bodied (faction - a description of people) with (behavior - a rule) (this is remembering for later):
    [ .. and so on ... ]
```

After Inform 7 expands the parameter *{invocation}* to *TryAction(v,w,x,y,z);* the pair of *{-backspace}* macros erase the semicolon and closing parenthesis so additional parameters could be added. Furthermore, we simply prepend the word "Could" in front of the function name to form a different function name altogether, *CouldTryAction()*, which we instructed to call the complicated to-phrase.

The last use of the (- and -) markers is to create a compile-time assertion, which Inform 7 calls "use options". To use the following we'd just assert it like any other use option: *Use the*

American dialect, RULES ALL at start, and no scoring.

Use RULES ALL at start translates as (- Global debug_rules=2; -).

The line will be included if the source includes the use option, otherwise not.

The other way of tinkering with Inform 6 is to expose a pre-existing variable, function, etc. by naming it with an Inform 7 name. Unlike the methods above, this allows us to assign to variables directly.

Out-of-world is a truth state that varies.
The out-of-world variable translates to I6 as "meta".

The currently-executing action is an action name that varies.
The currently-executing action variable translates into I6 as "action".

Properties, rules, understand tokens, *Definition:* adjectives, instantiations, and classes may be translated similarly. Translating is necessary because the generated I6 names differ from their I7 counterparts to prevent any unfortunate name clashes. But because I6 implements much of I7, tinkering requires communication between the layers. One example is new tokens for use in understand assertions, which inform the parser, written solely in Inform 6.

The understand token subordinating conjunction translates into I6 as "SUB_CONJ_TOKEN".

```
Include (-
[ SUB_CONJ_TOKEN;
        return ((+ parsing the sub conj +)-->1)();
];
-).
```

To decide which number is parse succeeds: (- GPR_PREPOSITION -).
To decide which number is parse fails: (- GPR_FAIL -).

```
To decide which number is parse the sub conj (this is parsing the sub conj):
        repeat through the table of subordinating conjunctions:
                if the unmatched word matches the topic entry:
                        now the sub conj is the output entry;
                        decide on parse succeeds;
        decide on parse fails.
```

We can declare global I6 variables from Inform 7, and not expose them to Inform 7 if we wish.

```
Include (-
Global save_debug_rules;
-).
```

When Inform 7 is creating Inform 6 code, it recognizes a few macros beside just *{-backspace}*. Although innocuous enough, these facilities open up whole new categories of

things phrases can do. Counters are the first. Each time a phrase uses *{-advance-counter:FOOBAR}*, the current value of the number variable FOOBAR replaces it as a constant literal, and FOOBAR is then post-incremented. The variable is inside the Inform 7 compiler, not in our source code (either I7 or I6). One example of its use is debugging. Let's say we have a complicated I6 inclusion that crashes with a runtime error, but only sometimes. We use the phrase in three different places in our code, and know not which is causing the problem. We could define it like so.

```
To do complicated thing:
(-      print "invocation #", {-advance-counter:BigThing}, "^";
        ! blah blah blah
 -).

When play begins, do complicated thing.
Every turn, do complicated thing.
```

Each place that "do complicated thing" appears in the source has its own number, starting from zero. We can also put the macro next to a variable name, like *myvar{-advance-counter:MyVarCount}*, so the resulting Inform 6 code references variables *myvar0*, *myvar1*, etc.

Related are *{-zero-counter:FOOBAR}* which sets the counter back to zero, and *{-counter:FOOBAR}* which becomes the number without increasing the number afterward. Just to be clear, *{-advance-counter:FOOBAR}* post-increments, and in case that causes a problem, it can be placed in an Inform 6 comment, where its expansion is ignored.

Finally in this series on counters is *{-allocate-storage:FOOBAR}*. The above counters only existed in our resulting Inform 6 source as constants, but this instruction creates actual storage space in our Inform 6 source, an array called *I7_ST_FOOBAR*, which has at least the number of elements as the FOOBAR counter. The counter is how each I7 phrase invocation knows which array element belongs to it, so *I7_ST_FOOBAR-->{-advance-counter:FOOBAR}* appears in our definition precisely once. Should we need to reference storage multiple times in the same definition, it appears last (because it post-increments) and one or more *I7_ST_FOOBAR-->{-counter:FOOBAR}* appear. For example, if our phrase defines a new kind of loop, and we want the loops to be able to nest, this construction solves our problem.

```
To repeat with (R - nonexisting rule variable) running through future events begin --
end:
(-
{-allocate-storage:LoopingThruEvents} ! expands to nothing, and no further effect for
multiple appearances
for ( I7_ST_LoopingThruEvents-->{-counter:LoopingThruEvents} = 1
  :  I7_ST_LoopingThruEvents-->{-counter:LoopingThruEvents} <=
TimedEventsTable-->0
  : (I7_ST_LoopingThruEvents-->{-counter:LoopingThruEvents})++ )
if (({R} = TimedEventsTable-->(I7_ST_LoopingThruEvents-->{-advance-
counter:LoopingThruEvents})) ~= 0)
-).                    [ that's an assignment inside the condition ]
```

And finally, almost as an afterthought: occasionally we write phrases for tables. Tables require some local variables, such as ct_1. The macro {-require-ctvs} tells Inform to create said variables.

This is only most of Inform 7. It is a large language with many nooks and crannies to explore. Finding them is frequently a game in itself. I hope you find it as enjoyable as I have.

READY.
> _