

Purpose

The global variables and those built-in rulebooks which do not belong either to specific actions or to specific activities.

A/sr2. §12-21 Rulebooks; §22 Rules; §23-24 Startup; §25-29 The turn sequence; §30 Shutdown; §31-32 Scene changing; §33-37 Action-processing; §38-41 Specific action-processing; §42 Player's action awareness; §43 Accessibility; §44 Reaching inside; §45 Reaching outside; §46 Visibility; §47-48 Does the player mean; §49 Adjectives applied to values

§1. With the kinds, objects and properties all now made, we turn to more abstract constructions: rules, rulebooks, activities, global variables and so on. As at the beginning of Part SR1 above, we begin with an entirely clean slate – none of any of these things exists yet – but also with expectations on what we do, since NI and the I7 template layer need certain constructions to be made.

We begin with global variables. The order in which these are defined, and the section subheadings under which they are grouped, determine the way they are indexed in the Contents index of a project – that is, moving them around would reorder the Contents index, and rewording the subheadings SR2/1, SR2/2, ..., below would change the text of the subheadings in the Contents index accordingly.

Some of these variables are special to NI, and it knows which they are not by the order in which they're defined (as with kinds, above) but by their (I7) names. These are marked with the comment `[*]`, and cannot be renamed without altering NI to match. Those marked `[**]` are similarly named in the template layer.

§2. It's now the occasion for our first global variable definition, because although users often think of the protagonist object as a fixed object whose name is "player", it's in fact possible to change perspective during play and become somebody else – at which point the "player" variable will point to a different object.

Note that "player" is actually a variable belonging to the I6 library, as indeed most of those in the Standard Rules are. Without the explicit translation supplied below, it would probably be compiled as `Q1_player` or some such name; and might well be implemented by NI as an entry in an array (since the Z-machine supports only a limited number of global variables, in effect using them as a set of 240 registers, and we therefore don't want to create too many ourselves).

Part SR2 - Variables and Rulebooks

Section SR2/1 - Situation

The player is a person that varies. `[*]`

The player variable translates into I6 as "player".

§3. The I7 variable “location” corresponds to I6’s `real_location`, not `location`. Its value is never equal to a pseudo-room representing darkness: it is always an actual room, and I7 has nothing corresponding to I6’s `thedark` “room”. Similarly, we use an I7 variable “darkness witnessed” for a flag which the I6 library would have stored as the `visited` attribute for the `thedark` object.

The “maximum score” is, rather cheekily, translated to an I6 constant: and this cannot be changed at run-time.

The location -- documented at `var_location` -- is an object that varies. [*]

The score -- documented at `var_score` -- is a number that varies.

The last notified score is a number that varies.

The maximum score is a number that varies. [*]

The turn count is a number that varies.

The time of day -- documented at `var_time` -- is a time that varies. [*]

The darkness witnessed is a truth state that varies.

The location variable translates into I6 as `"real_location"`.

The score variable translates into I6 as `"score"`.

The last notified score variable translates into I6 as `"last_score"`.

The maximum score variable translates into I6 as `"MAX_SCORE"`.

The turn count variable translates into I6 as `"turns"`.

The time of day variable translates into I6 as `"the_time"`.

§4. It is arguable that “noun”, “second noun” and “person asked” ought to be rulebook variables belonging to the action-processing rules, so that they essentially did not exist outside of the context of an ongoing action. The reason this isn’t done is partly historical (rulebook variables were a fairly late development, implemented in April 2007, though they had long been planned). But it is sometimes useful to get at the nouns in an action after it has finished, and making them global variables also makes them a little faster to look up (a good thing since they are used so much), and causes them to be indexed more prominently.

“Item described” is simply an I7 name for `self`. (In an object-oriented system such as I6, `self` is the natural way to refer to the object currently under discussion, within routines applying to all objects of its class.) In early drafts of I7, it was called “this object”, but somehow this raised expectations too high about how often it would be meaningful: it looked like a pronoun running meanings from sentence to sentence.

Section SR2/2 - Current action

The noun -- documented at `var_noun` -- is an object that varies. [*]

The second noun is an object that varies. [*]

The person asked -- documented at `var_person_asked` -- is an object that varies. [*]

The reason the action failed -- documented at `var_reason` -- is a rule that varies.

The item described is an object that varies.

The noun variable translates into I6 as `"noun"`.

The second noun variable translates into I6 as `"second"`.

The person asked variable translates into I6 as `"actor"`.

The reason the action failed variable translates into I6 as `"reason_the_action_failed"`.

The item described variable translates into I6 as `"self"`.

§5. “Person reaching” turns out to have exactly the same meaning as “person asked” – they are both the actor, in I6 terms, but are used in different situations.

Section SR2/3 - Used when ruling on accessibility

The person reaching -- documented at var_person_reaching -- is an object that varies.

The container in question is an object that varies.

The supporter in question is an object that varies.

The particular possession -- documented at var_particular -- is a thing that varies.

The person reaching variable translates into I6 as "actor".

The container in question variable translates into I6 as "parameter_object".

The supporter in question variable translates into I6 as "parameter_object".

The particular possession variable translates into I6 as "particular_possession".

§6. Parsing variables follow. The I6 parser tends to put any data read as part of a command into the variable `parsed_number`, but then, I6 is typeless: we can't have a single I7 variable for all these possibilities since it could then have no legal type. We solve this as follows. Whenever a kind of value *K* is created which can be parsed as part of a command, an I7 variable “the *K* understood” is also created, as a *K* that varies. All of these variables are translated into I6's `parsed_number`, so in effect they provide aliases of each possible type for the same underlying memory location. The four exceptional kinds of value not parsed by the systematic approaches in NI for enumerated KOVs and units are “number”, “time”, “snippet” and “truth state”: because of their exceptional status, they don't get “the *K* understood” variables created automatically for them, so we must construct those by hand. Hence “the number understood”, “the time understood”, “the topic understood” (for historical reasons this one is not called “the snippet understood”), “the truth state understood” but no others.

Section SR2/4 - Used when understanding typed commands

The player's command -- documented at var_command -- is a snippet that varies.

The matched text is a snippet that varies.

The number understood -- documented at var_understood -- is a number that varies. [*]

The time understood is a time that varies. [*]

The topic understood is a snippet that varies. [*]

The truth state understood is a truth state that varies. [*]

The current item from the multiple object list is an object that varies.

The player's command variable translates into I6 as "players_command".

The matched text variable translates into I6 as "matched_text".

The topic understood variable translates into I6 as "parsed_number".

The truth state understood variable translates into I6 as "parsed_number".

The current item from the multiple object list variable translates into I6 as "multiple_object_item".

§7. The following are, unless the user creates global variables of his own, the last in the Contents index...

Section SR2/5 - Presentation on screen

The command prompt -- documented at var_prompt -- is a text that varies. [**]

The command prompt is ">".

The left hand status line -- documented at var_sl -- is a text that varies.

The right hand status line is a text that varies.

The left hand status line variable translates into I6 as "left_hand_status_line".

The right hand status line variable translates into I6 as "right_hand_status_line".

The listing group size is a number that varies.

The listing group size variable translates into I6 as "listing_size".

§8. ...but they are not the last global variables created by the Standard Rules.

These bibliographic data variables are concealed because they are under a heading which ends with the word “unindexed”. There are two reasons why these variables are unindexed: first, they appear in a different guise only a little lower in the Contents index as part of the Library Card, and second, we don’t want users to think of them as manipulable during play.

Rather sneakily, we also define a Figure here. This is done in order to make it legal to declare variables and properties of the kind of value “figure-name” (because it ensures that such variables can always be initialised – there is always at least one Figure in the world). Of course plenty of Inform projects have no artwork at all: so the cover art figure is unique in that it might refer to nothing. That sounds a little arbitrary but in fact follows a convention used by the Blorb format for binding story files with their resources, which in turn follows Infocom conventions of 1987-89: the cover art always has resource ID number 1, whether it exists or not. NI creates figures and sound effects counting upwards from 1, giving them each unique resource ID numbers, so the first to be created gets ID number 1: by defining “figure of cover” here, we can be sure that we are first, so everything works.

Section SR2/6 - Unindexed Standard Rules variables - Unindexed

The story title, the story author, the story headline, the story genre and the story description are text variables. [*****]

The release number and the story creation year are number variables. [**]

Figure of cover is the file of cover art.

The story title variable translates into I6 as "Story".

§9. And we finish out the set with some “secret” variables used only by the Standard Rules or by NI, and which are therefore also unindexed. Their names are all hyphenated, to reduce the chance of anyone stumbling into them in a namespace clash.

The first set of three secret variables is used by the predicate calculus machinery in NI. This is the code which handles logical sentences such as “at least six doors are open” or descriptions such as “open doors”, by reducing them to a logical notation which sometimes makes use of variables. For instance, “open doors” reduces to something like “all x such that $\text{door}(x)$ and $\text{open}(x)$ ”, with x being a variable. When NI works with such logical propositions, it often needs to substitute $x = c$, that is, to replace x with a given constant c . But it can only do this if c is an Inform 7 value. This is a problem if what it wants is to substitute in something which is only meaningful at the I6 level: say, it wants to substitute a value c which will eventually translate to *whatever*, but it can’t find any I7 value c which will do this.

We solve this problem by constructing some unusual I7 variables whose only purpose is that NI can use them in substitutions. They should never be referred to in I7 source text anywhere else at all, not even elsewhere in the Standard Rules.

- (1) The “substitution-variable” is used when NI needs to compile a proposition with one free variable, such as “ $\text{door}(x)$ ”, as a condition: it binds x with the substitution $x = \text{say-parameter}$ in order to turn this into a well-formed predicate calculus sentence with no free variables, then compiles it. This is a trick used in the definition of some of the “repeat” phrases below, as a result of the `{-bind-variable:...}` invocation command.
- (2) The sentence “The i6-nothing-constant is an object that varies.” is a rare example of a flat lie in the Standard Rules, as it is the I6 constant `nothing` and never varies at all. Again, it exists as a “variable” so that the substitution $x = \text{nothing}$ can be made.
- (3) Well, once you start telling lies it’s so hard to stop, and it’s also a lie that the “I6-varying-global” translates to `nothing`. It actually translates to whatever the NI machinery for compiling propositions happens to want at the moment, so it has no permanent meaning at all. (It will always translate to an I6 global variable storing a value whose I7 kind is “object”, so the type-checking machinery isn’t endangered by this chicanery. It will in fact never translate to `nothing`, but we make the translation sentence below in order to avoid allocating any storage at run-time for what is in the end only a label.)

The substitution-variable is an object that varies. [*]
 The substitution-variable variable translates into I6 as "subst_v".
 The I6-nothing-constant is an object that varies. [*]
 The I6-nothing-constant variable translates into I6 as "nothing".
 The I6-varying-global is an object that varies. [*]
 The I6-varying-global variable translates into I6 as "nothing".

§10. The remaining secret variables are:

- (1) The “item-pushed-between-rooms” is needed to get the identity of an object being pushed by a command like PUSH ARMCHAIR NORTH out of I6 and into the action variable “thing gone with” of the going action.
- (2) The “actor-location” is needed temporarily to store the room in which the actor of the current action is standing, and it wants to be an I6 global (rather than, say, a rulebook variable belonging to the action-processing rulebook) so that NI can use common code to handle this alongside noun, second and actor when compiling preambles to rules.
- (3) The “parameter-object” is likewise needed in order to compile preambles to rules in object-based rulebooks.

The item-pushed-between-rooms is an object that varies.
 The item-pushed-between-rooms variable translates into I6 as "move_pushing".
 The actor-location is an object that varies. [*]
 The actor-location variable translates into I6 as "actor_location".
 The parameter-object is an object that varies. [*]
 The parameter-object variable translates into I6 as "parameter_object".

§11. And that completes the run through all the variables created in the Standard Rules.

§12. Rulebooks. There are 25 rulebooks which are, so to speak, “primitive” in Inform 7 – which are part of its workings and cannot safely be tampered with. NI requires them to be declared by the Standard Rules as the first 25 rulebooks to be created, and in the exact order below. (In fact, though, this is mostly so that it can prepare the index pages correctly: NI is not the part of I7 which decides what to use these rulebooks for. That is almost always the responsibility of the template I6 layer which, for instance, calls upon the action-processing rulebook when it wants an action processed.)

At the I6 level, a rulebook is referred to by its ID number, which counts upwards from 0 in order of creation. Any reordering of the constants below, therefore, is unsafe unless changes are made elsewhere so that the following three tallies always remain in synchrony:

- (a) The sequence of declaration of these rulebooks in the Standard Rules.
- (b) The inweb `@d` definitions in the form `TURN_SEQUENCE_RB` in the section Rulebooks of Chapter 12 of the NI source code.
- (c) The I6 `Constant` definitions in the form `TURN_SEQUENCE_RB` in the file `Rulebooks.i6t` of the template I6 layer.

Anyway, we will declare the rulebooks and their variables or outcomes first, and come back to stock some of them with rules later. It seems appropriate to give first place to the procedural rules, the super-rulebook capable of controlling all others:

Section SR2/7 - The Standard Rulebooks

Procedural rules is a rulebook. [0]

§13. Every story file created by Inform 6 begins execution in a routine called `Main`, which is analogous to the `main()` function of a C program: it is as if the entire program is a call to this function.

In an I7 story file, the code in `Main` is the only code which is not executed in the context of a rulebook, and by design it does as little as possible. The definition is in part “Main” of “OrderOfPlay.i6t”, and this is what it does:

- (1) Consider the startup rules.
- (2) Repeatedly follow the turn sequence rules until `deadflag` is set, which is an I6 variable used to indicate that the game has ended in one way or another.
- (3) Follow the shutdown rules.

The startup rules are only considered, not followed, because we cannot risk a procedural rule being the first thing executed: starting up the virtual machine correctly has to be our first priority. (Except for not using the procedural rules, considering a rulebook is identical to following it.)

Briefly, the startup phase takes us to the end of the room description after the banner is printed. The turn sequence covers a complete turn, and runs through from prompting the player for a command to notifying him of any change in score which occurred. The shutdown rules then go from printing the obituary text, through final score, to the question about quitting or restarting.

Startup rules is a rulebook. [1]

Turn sequence rules is a rulebook. [2]

Shutdown rules is a rulebook. [3]

§14. Now a set of rulebooks to do with the passage of time.

Scene changing rules is a rulebook. [4]

When play begins is a rulebook. [5]

When play ends is a rulebook. [6]

Every turn rules is a rulebook. [7]

§15. The action machinery requires some 16 rulebooks to work, though that is the result of gradual simplification – in 2006 it required 25, for instance. The “action-processing” rulebook, like the turn sequence rulebook, is a master of ceremonies: it belongs to the Standard Rules and is only rarely if at all referred to by users.

As remarked above, it’s something of a historical accident that “actor” is a rulebook variable belonging to the action-processing rules (and thus in scope for every rulebook it employs) while “noun” and “second noun” are global variables (and thus in scope everywhere).

The main action-processing rulebook delegates most of its detailed work to a subsidiary, the “specific action-processing” rulebook, at the point where what rulebooks we consult next depends on what the action is (hence “specific”) – see below for more on how check/carry out/report rules are filed.

```
Action-processing rules is a rulebook. [8]
The action-processing rulebook has a person called the actor.
Setting action variables is a rulebook. [9]
The specific action-processing rules is a rulebook. [10]
The specific action-processing rulebook has a truth state called action in world.
The specific action-processing rulebook has a truth state called action keeping silent.
The specific action-processing rulebook has a rulebook called specific check rulebook.
The specific action-processing rulebook has a rulebook called specific carry out rulebook.
The specific action-processing rulebook has a rulebook called specific report rulebook.
The specific action-processing rulebook has a truth state called within the player’s sight.
The player’s action awareness rules is a rulebook. [11]
```

§16. The rules on accessibility and visibility, which control whether an action is physically possible, have named outcomes as a taste of syntactic sugar.

```
Accessibility rules is a rulebook. [12]
Reaching inside rules is an object-based rulebook. [13]
Reaching inside rules have outcomes allow access (success) and deny access (failure).
Reaching outside rules is an object-based rulebook. [14]
Reaching outside rules have outcomes allow access (success) and deny access (failure).
Visibility rules is a rulebook. [15]
Visibility rules have outcomes there is sufficient light (failure) and there is
insufficient light (success).
```

§17. Two rulebooks govern the processing of asking other people to carry out actions:

```
Persuasion rules is a rulebook. [16]
Persuasion rules have outcomes persuasion succeeds (success) and persuasion fails (failure).
Unsuccessful attempt by is a rulebook. [17]
```

§18. Next, the six classic rulebooks best known to users of Inform. It's perhaps an unfortunate point of the design that there are so many as six: that seems rather a lot of stages to go through, and indeed means that there is sometimes some ambiguity about which rulebook to use if one wants to achieve a given effect. There are really two reasons why things are done this way:

- (a) To try to encourage a distinction between:
 - (i) the general implementation of an action, made with carry out, check and report rules – say, a “photographing” action which could be used in any situation and could be copied and pasted into another project; and
 - (ii) the contingent rules applying in particular situations in play, made with before, instead and after rules, such as that custodians at the Metropolitan Museum of Art forbid flash photography.
- (b) To improve the efficiency of action-processing by forcing control to run only through those carry out, check and report rules which can possibly be relevant to the current action. Whereas all before, instead and after rules are all piled up together in their own rulebooks, check, carry out and report rules are divided up into specialised rulebooks tied to the particular action in progress. Thus on a taking action, the six stages followed are before, instead, check taking, carry out taking, after and report taking.

During play, then, the three rulebooks “check”, “after” and “report” are completely empty. This is the result of a reform in April 2007 which wasn't altogether popular. Before then, NI rather cleverly filed rules like “Check doing something with the haddock” in the generic “check” rulebook and ran this rulebook as part of the action processing sequence. But this clearly broke principle (i) above, and meant that the six-stage process – already quite complicated enough – was actually a nine-stage process only pretending, by deceitful syntax, to be a six-stage one. Check rules sometimes appeared to be filed in the wrong order, breaking the ordinary precedence conventions, and this was not due to a bug but because they were only pretending all to be in the same rulebook. Still more clever indexing and rule-ordering tricks ameliorated this a little, but in the end it was just a bad design: withdrawing the ability to make check, carry out and report rules apply to multiple actions resulted in much cleaner code, and also a clearer conceptual definition of what these rulebooks were for. (But users still *didn't* like the change: actual functionality was withdrawn.)

So if they are always empty and never used, why are the three rulebooks called simply “check”, “after” and “report” created in the first place? The answer is that this is a convenience for parsing rule preambles in NI: it provides a temporary home for such rules before they are divided up into their specific rulebooks, and it also makes it easier for NI to detect and give a helpful Problem message in response to rules like “Check taking or dropping the perch” which can't be filed anywhere in our scheme, and so have to be forbidden.

Before rules is a rulebook. [18]

Instead rules is a rulebook. [19]

Check rules is a rulebook. [20]

Carry out rules is a rulebook. [21]

After rules is a rulebook. [22]

Report rules is a rulebook. [23]

§19. The final rulebook in this roundup is one used for parsing ambiguous commands during play. This looks like the job of an activity rather than a rulebook, but (i) we want information in the form of a nicely-named five point scale of responses, rather than wanting something to be done, and (ii) it doesn't really make any sense to split that question into a before, for and after stage. So this is a named rulebook instead.

The does the player mean rules are a rulebook. [24]

The does the player mean rules have outcomes it is very likely, it is likely, it is possible, it is unlikely and it is very unlikely.

§20. “Does the player mean” is essentially a front end for the I6 parser’s `ChooseObjects` entry point, which relies on numerical scores to assess the likelihood of possible choices. That makes it useful to have an I6 wrapper function which consults the “does the player mean” rulebook, translates the named outcomes into a score from 0 to 4, and returns that. Note that if the rulebook makes no decision and has no outcome, we return the middle-of-the-road value 2. (This wrapper routine looks as if it belongs in the template I6 layer, but having it here allows it to use the (+ and +) escapes.)

```

Include (-
  [ CheckDPMR result sinp1 sinp2 rv;
    sinp1 = inp1; sinp2 = inp2; inp1 = noun; inp2 = second;
    rv = FollowRulebook( (+does the player mean rules+) );
    inp1 = sinp1; inp2 = sinp2;
    if ((rv) && RulebookSucceeded()) {
      result = ResultOfRule();
      if (result == (+ it is very likely outcome +) ) return 4;
      if (result == (+ it is likely outcome +) ) return 3;
      if (result == (+ it is possible outcome +) ) return 2;
      if (result == (+ it is unlikely outcome +) ) return 1;
      if (result == (+ it is very unlikely outcome +) ) return 0;
    }
    return 2;
  ];
-);

```

§21. And that’s it: all 25 of the named rulebooks now exist. There will, of course, be hundreds more rulebooks soon, created automatically as activities and actions are created – when we create the “dropping” action, for instance, we also create the “check dropping”, “carry out dropping” and “report dropping” rulebooks – but there are no more stand-alone rulebooks.

§22. **Rules.** At run-time, the value of a rule is the (packed) address of an I6 routine. In the case of rules created in I7 source text, the body of the rule definition is compiled by NI to a new I6 routine which carries it out. But there are also primitive rules which are implemented in the template I6 layer, and these need no I7 source text definitions: instead we simply tell NI the name of the I6 routine which will be handling this rule, and that it need not bother to create one for itself.

An example of this is provided by the first rule we shall create: the “little-used do nothing rule”. This is aptly named on both counts. We never follow it, we never put it into any rulebook. It exists only so that variables and properties with the kind of value “rule” can be initialised automatically to a safely neutral default value. It makes no decision.

Section SR2/8 - The Standard Rules

The little-used do nothing rule translates into I6 as "LITTLE_USED_DO_NOTHING_R".

```

Include (-
  [ LITTLE_USED_DO_NOTHING_R; rfalse; ];
-);

```

§23. Startup. The startup rulebook is considered but not followed (see above) and so is immunised from the effect of procedural rules.

Every rulebook contains a (possibly empty) run of “first” rules, then a (possibly empty) run of miscellaneous rules, then a (possibly empty) run of “last” rules. It’s unusual to have more than one rule anchored to either end as “first” or “last”, but entirely legal, and we make use of that ability here.

The “first” rules here are the ones which get the basic machinery working to the point where it is safe to run arbitrary I7 source text. They necessarily do very low-level things, and it is not guaranteed that I7 phrases will behave to specification if executed before these early rules have finished. So it is hazardous to obstruct or alter them.

- (a) The “virtual machine startup rule” carries out necessary steps to begin execution on the virtual machine in use: this entails relatively little on the Z-machine versions 5 or 8, but can involve extensive work to get the screen display working on Glulx or Z6. Before anything else happens, however, the “starting the virtual machine” activity (see below) is carried out: again, this is done in a careful way which avoids procedural rules firing.
- (b) The “initialise memory rule” starts up the memory allocation heap, if there is one, and sets some essential I6 variables. If there is any rule not to meddle with, this is it.
- (c) The “seed random number generator rule” seeds the RNG to a fixed value if NI has requested this (which it does in response to the `-rng` command line switch, which is in turn used by the `intest` testing utility: it’s a way to make deterministic tests of programs which use random values).
- (d) The “update chronological records rule” is described in further detail below, since it appears both here and also in the turn sequence rulebook. Here it’s providing us with a baseline of initial truths from which we can later assess conditions such as “the marble door has been open”. A subtle and questionable point of the design is that this rule is placed at a time when the object representing the player is not present in the model world. This is done to avoid regarding the initial situation as one of the turns for purposes of a rule preamble like “... when the player has been in the Dining Room for three turns”. It’s as if the player teleports into an already-existing world, like some Star Trek crewman, just in time for the first command.
- (e) And the “position player in model world rule” completes the initial construction of the spatial model world.
- (f) The “start in the correct scenes rule” ensures that we start out in the correct scenes. (This can’t wait, because it’s just conceivable that somebody has written a rule with a preamble like “When play begins during the Hunting Season...”: it’s also where the scene Entire Game begins.) That completes the necessary preliminaries before ordinary I7 rules can be run.

The start in the correct scenes rule is listed first in the startup rulebook. [6th.]

The position player in model world rule is listed first in the startup rulebook. [5th.]

The update chronological records rule is listed first in the startup rulebook. [4th.]

The seed random number generator rule is listed first in the startup rulebook. [3rd.]

The initialise memory rule is listed first in the startup rulebook. [2nd.]

The virtual machine startup rule is listed first in the startup rulebook. [1st.]

The virtual machine startup rule translates into I6 as "VIRTUAL_MACHINE_STARTUP_R".

The initialise memory rule translates into I6 as "INITIALISE_MEMORY_R".

The seed random number generator rule translates into I6 as "SEED_RANDOM_NUMBER_GENERATOR_R".

The update chronological records rule translates into I6 as "UPDATE_CHRONOLOGICAL_RECORDS_R".

The position player in model world rule translates into I6 as "POSITION_PLAYER_IN_MODEL_R".

This is the start in the correct scenes rule: consider the scene changing rules.

§24. The remaining rules, though, are fair game for alteration, and as if to prove the point they are all written in standard I7 source text. Note that the baseline score is set only after the when play begins rulebook has finished, because games starting with a non-zero score normally do this by changing the score in a when play begins rule: and we don't want such a change to be notified to the player as if it has happened through some action.

The when play begins stage rule is listed in the startup rulebook.

The fix baseline scoring rule is listed in the startup rulebook.

The display banner rule is listed in the startup rulebook.

The initial room description rule is listed in the startup rulebook.

This is the when play begins stage rule: follow the when play begins rulebook.

This is the fix baseline scoring rule: now the last notified score is the score.

This is the display banner rule: say "[banner text]".

This is the initial room description rule: try looking.

§25. **The turn sequence.** In each turn, a command is read and parsed from the keyboard, and any action(s) that requested is or are processed. (And may in turn cause other actions, which must also be processed.) There is then a fair amount of business needed to end one turn and get ready for another.

The turn sequences rulebook terminates early if `deadflag` becomes set at any point, so the last turn of play will be incomplete. Besides that consideration, it can also end early if the command for the turn was for an out-of-world action such as saving the game: in such cases, the “generate action rule” stops the rulebook once the action has been fully processed. All the same, play basically consists of running down this list of rules over and over again.

The turn sequence rulebook is the only one protected from being ignored by a procedural rule: ignoring it would lock the story file into an endless loop, so a run-time problem is issued instead if this is tried. (In pre-2008 drafts of Inform 7, ignoring what was then the turn sequence rulebook was a legitimate tactic, so the RTP message explains how to achieve the same effect by new methods.)

§26. The “first” rules in the turn sequence cover us up to the end of the events which take place in the model world during this turn's action(s).

- (a) The “parse command rule” prints up the prompt, reads a command from the keyboard, parses it into dictionary words, deals with niceties such as UNDO or OOPS, and then runs it through the traditional I6 parser to turn it into a request for an action or list of actions. But see note below.
- (b) The “generate action rule” then either sends a single action to the action-processing rules, or else runs through the list, printing the noun up with a colon each time and then sending the action to the action-processing rules. But see note below.
- (c) We then run the scene changing rulebook, because people often write every turn rules which are predicated on particular scenes (“Every turn during the Grand Waltz: ...”), and such rules will fail if we haven't kept up with possible scene changes arising from something done in the action(s) just completed.
- (d) The “every turn stage rule” follows the every turn rulebook. This earns its place among the “first” rules in order for it to have priority over all the other book-keeping rules at the end of a turn – including any which the user, or an extension included by the user, chooses to add to the turn sequence rules.

An unusual point here is that the “parse command rule” and the “generate action rule” are written such that they *do nothing unless the turn sequence rulebook is being followed at the top level* (by `Main`, that is). This prevents them from being used recursively, which would not work properly, and enables a popular trick from the time before the 2008 reform to keep working: we can simulate six turns going by in which the player does nothing by running “follow the turn sequence rules” six times in a row. Everything happens exactly as it should – the turn count, the time of day, timed events, and so on – except that no commands are read and no consequent actions generated.

A first turn sequence rule (this is the every turn stage rule): follow the every turn rules. [4th.]
 A first turn sequence rule: consider the scene changing rules. [3rd.]
 The generate action rule is listed first in the turn sequence rulebook. [2nd.]
 The parse command rule is listed first in the turn sequence rulebook. [1st.]

§27. Three miscellaneous things then happen, all implemented by primitives in the template I6 layer:

- (e) The “timed events rule” is the one which causes other rules, keyed to particular times of day, to fire.
- (f) The “advance time rule” then causes the “time of day” global variable to advance by the duration of a single turn, which by default is 1 minute.
- (g) The “update chronological records rule” tells the chronology machine that a slice of time has been completed. Inform can only decide past tense conditions like “the black door has been open” by continuously measuring the present to see if the black door is open now, and making a note for future reference if it is. But of course it’s impossible for any computer to continuously do anything, and besides, Inform has other calls on it. What in fact happens is the Inform performs these measurements at “chronology points”, which are strategic moments during play, and this is one of them.

The timed events rule is listed in the turn sequence rulebook.
 The advance time rule is listed in the turn sequence rulebook.
 The update chronological records rule is listed in the turn sequence rulebook.

§28. We now come to the rules anchored at the end, using “last”. This part of the rulebook is reserved for book-keeping which has to happen positively at the end of the turn.

- (h) First, we check for scene changes again. We did this only a short while ago, but scene changes might well have arisen as a result of rules which fired during the every turn rulebook, or from timed events, or in some other way, and it’s important to start the next turn in the correct scene – so we check again to make sure.
- (i) Then we run the “adjust light rule”. Keeping track of light and darkness is quite difficult, and potentially also quite slow: it’s not unlike a sort of discretised version of ray-tracing, with many light sources and barriers to think about (some transparent, some opaque). So we do this as little as possible: once per turn we calculate whether the player is in light or not, and act accordingly if so.
- (j) The “note object acquisitions rule” does two things:
 - (i) Gives the “handled” property to everything carried or worn by the player.
 - (ii) Changes the current player’s holdall in use, if necessary. (That’s to say: if the player has dropped his previous player’s holdall, we try to find a new one to use from his remaining possessions.)
- (k) The “notify score changes rule” tells the player if the score has changed during the turn, or rather, since the last time either this rule or the startup “fix baseline scoring rule” ran. (If the score were to change in the course of an out-of-world action, it would be notified a turn late, but of course out-of-world actions are not supposed to do that sort of thing.)

A last turn sequence rule: consider the scene changing rules. [3rd from last.]
 The adjust light rule is listed last in the turn sequence rulebook. [2nd from last.]
 The note object acquisitions rule is listed last in the turn sequence rulebook. [Penultimate.]
 The notify score changes rule is listed last in the turn sequence rulebook. [Last.]

This is the notify score changes rule:

```
if the score is not the last notified score:
    issue score notification message;
    now the last notified score is the score;
```

§29. That's it, but we need to map these I7 rule names onto the names of their I6 primitives in the template layer.

The `adjust light` rule translates into I6 as `"ADJUST_LIGHT_R"`.

The `advance time` rule translates into I6 as `"ADVANCE_TIME_R"`.

The `generate action` rule translates into I6 as `"GENERATE_ACTION_R"`.

The `note object acquisitions` rule translates into I6 as `"NOTE_OBJECT_ACQUISITIONS_R"`.

The `parse command` rule translates into I6 as `"PARSE_COMMAND_R"`.

The `timed events` rule translates into I6 as `"TIMED_EVENTS_R"`.

§30. **Shutdown.** Goodbye is not the hardest word to say, but it does involve a little bit of work. It might not actually be goodbye, for one thing: if this rulebook ends in success, then we go back to repeating the turn sequence rulebook just as if nothing had happened.

- (a) The “when play ends stage rule” follows the rulebook of the same name.
- (b) The “resurrect player if asked rule” does nothing unless one of the “when play ends” rules ran the “resume the game” phrase, in which case it stops the rulebook with success (see above).
- (c) The “print player’s obituary rule” carries out the activity of nearly the same name (see below).
- (d) The “ask the final question rule” asks the celebrated “Would you like to RESTART, RESTORE a saved game or QUIT?” question, and acts on the consequences. It can also cause an UNDO, and on a victorious ending may carry out the “amusing a victorious player” activity (see below). The rule only actually ends in the event of a QUIT: an UNDO, RESTART or RESTORE is performed at the hardware level by the virtual machine and destroys the current execution context entirely.

The `when play ends stage` rule is listed first in the shutdown rulebook.

The `resurrect player if asked` rule is listed last in the shutdown rulebook.

The `print player’s obituary` rule is listed last in the shutdown rulebook.

The `ask the final question` rule is listed last in the shutdown rulebook.

This is the `when play ends stage` rule: follow the `when play ends` rulebook.

This is the `print player’s obituary` rule:

carry out the printing the player’s obituary activity.

The `resurrect player if asked` rule translates into I6 as `"RESURRECT_PLAYER_IF_ASKED_R"`.

The `ask the final question` rule translates into I6 as `"ASK_FINAL_QUESTION_R"`.

§31. **Scene changing.** Scene changing is handled by a routine called `DetectSceneChange` which is compiled directly by NI: this is so primitive that it can’t even be handled at the template layer. The rulebook is all a little elaborate given that it contains only one rule, but it’s possible to imagine extensions which need to do book-keeping similar to scene changing and which want to make use of this opportunity.

The `scene change machinery` rule is listed last in the scene changing rulebook.

The `scene change machinery` rule translates into I6 as `"DetectSceneChange"`.

§32. We couldn’t do this earlier (because creating a scene automatically generates two rulebooks, and that would have thrown the rulebook numbering), so let’s take this opportunity to define the “Entire Game” scene:

Section SR2/9 - The Entire Game scene

The Entire Game is a scene.

The Entire Game begins when the game is in progress.

The Entire Game ends when the game is over.

§33. Action-processing. Action-processing happens on two levels: an upper level, handled by the main “action-processing” rulebook, and a lower level, “specific action-processing”. This division clearly complicates matters, so why do we do it? It turns out to be convenient for several reasons:

- (a) Out-of-world actions like “saving the game” need to run through the lower level, or they won’t do anything at all, but must not run through the upper level, or in-world rules (before or instead rules, for instance) might prevent them from happening.
- (b) Requested actions such as generated by a command like “CLARK, BLOW WHISTLE” have the reverse behaviour, being handled at the upper level but not the lower. (If Clark should agree, a definite non-request action “Clark blowing the whistle” is generated afresh: that one does indeed get to the lower level, but the original request action doesn’t.)
- (c) Specific action-processing has a rather complicated range of outcomes: it must succeed or fail, according to whether the action either reaches the carry out rules or is converted into another action which does, but also ensure that in the event of failure, the exact rule causing the failure is recorded in the “reason the action failed” variable.
- (d) The specific action-processing stage is where we have to split consideration into action-specific rulebooks (like “check taking”) rather than general ones (like “instead”). To get this right, we want to use some rulebook variables, and these need to be set at exactly the correct moment. It’s tricky to arrange for that in the middle of a big rulebook, but easy to do so at the beginning, so we want the start of the SA-P stage to happen at the start of a rulebook.

This does mean that an attempt by the user to move the before stage to just after the check stage (say) will fail – the before and check stages happen in different rulebooks, so no amount of rearranging will do this. (Procedural rules could still achieve this very doubtfully wise effect in any case.)

§34. The upper level of action-processing consists of seeing whether the actor’s current situation forbids the action from being tried: does anybody or anything intervene to stop it? Are there any basic reasons of physical realism why nobody could possibly try it? Does it require somebody else to cooperate who in fact chooses not to?

Doctrinally, the before stage must see actions before anything else can happen. (It needs the absolute freedom to start fresh actions and dispose of the one originally intended, sure in the knowledge that no other rules have been there first.) So before-ish material is anchored at the “first” end of the rulebook.

The other book-end on the shelf is provided by the instead stage, which doctrinally happens only when the action is now thought to be physically reasonable. So this is anchored at the “last” end. (In fact it is followed by several more rules also anchored there, but this is essentially just the clockwork machinery showing through: they aren’t realism checks.)

Miscellaneous general rules to do with physical realism are placed between the two book-ends, then, and this is where any newly created action-processing rules created by the user or by extensions will go.

Section SR2/10 - Action processing

The before stage rule is listed first in the action-processing rules. [3rd.]

The set pronouns from items from multiple object lists rule is listed first in the action-processing rules. [2nd.]

The announce items from multiple object lists rule is listed first in the action-processing rules. [1st.]

The basic visibility rule is listed in the action-processing rules.

The basic accessibility rule is listed in the action-processing rules.

The carrying requirements rule is listed in the action-processing rules.

The instead stage rule is listed last in the action-processing rules. [4th from last.]

The requested actions require persuasion rule is listed last in the action-processing rules.

The carry out requested actions rule is listed last in the action-processing rules.

The descend to specific action-processing rule is listed last in the action-processing rules.

The end action-processing in success rule is listed last in the action-processing rules. [Last.]

§35. As we shall see, most of these rules are primitives implemented by the template I6 layer, but five are very straightforward.

```
This is the set pronouns from items from multiple object lists rule:
    if the current item from the multiple object list is not nothing,
        set pronouns from the current item from the multiple object list.
```

```
This is the announce items from multiple object lists rule:
    if the current item from the multiple object list is not nothing,
        say "[current item from the multiple object list]: [run paragraph on]".
```

```
This is the before stage rule: abide by the before rules.
```

```
This is the instead stage rule: abide by the instead rules.
```

§36. The final rule in the rulebook always succeeds: this ensures that action-processing always makes a decision. (I7's doctrine is that an action "succeeds" if and only if its carry-out stage is reached, so any action getting right to the end of this rulebook must have succeeded.)

```
This is the end action-processing in success rule: rule succeeds.
```

§37. The action-processing rulebook contains six primitives:

- (1) The "basic visibility rule" checks the action to see if it requires light, and if so, and if the actor is the player and in darkness, asks the visibility rules (see below) whether the lack of light should stop the action. (It would be cleaner to apply this rule to all actors, but we would need much more extensive light calculations to do this.)
- (2) The "basic accessibility rule" checks the action to see if it requires the noun to be touchable, and if so, asks the accessibility rulebook to adjudicate (see below); then repeats the process for the second noun.
- (3) The "carrying requirements rule" checks the action to see if it requires the noun to be carried. If so, but the noun is not carried, it generates an implicit taking action (in effect, "try silently taking *N*"); and if that fails, then the rulebook is halted in failure. The process is then repeated for the second noun.
- (4) If the action is one where the player requests somebody else to do something, the "requested actions require persuasion rule" asks the persuasion rulebook for permission.
- (5) If the action is one where the player requests somebody else to do something, the "carry out requested actions rule" starts a new action by the person asked, and looks at the result: if it failed, the "unsuccessful attempt by" rulebook is run to tell the player what has (not) happened. (If that does nothing, a bland message is printed.) In all cases, the original action of requesting is ended in success: a success because, whatever happened, the request succeeded in making the actor try to do something.
- (6) The "descend to specific action-processing rule" really only runs the specific action-processing rulebook, but it's implemented as a primitive in the template I6 layer because it must also find out which are the specific check, carry out and report rulebooks for the current action (for instance, "check taking" is the specific check rulebook for the "taking" action – which seems obvious from the names: but at run-time, the names aren't so visible).

```
The basic accessibility rule translates into I6 as "BASIC_ACCESSIBILITY_R".
```

```
The basic visibility rule translates into I6 as "BASIC_VISIBILITY_R".
```

```
The carrying requirements rule translates into I6 as "CARRYING_REQUIREMENTS_R".
```

```
The requested actions require persuasion rule translates into I6 as "REQUESTED_ACTIONS_REQUIRE_R".
```

```
The carry out requested actions rule translates into I6 as "CARRY_OUT_REQUESTED_ACTIONS_R".
```

```
The descend to specific action-processing rule translates into I6 as
"DESCEND_TO_SPECIFIC_ACTION_R".
```

§38. **Specific action-processing.** And now we descend to the lower level, which is much easier to understand.

The work out details of specific action rule is listed first in the specific action-processing rules.

A specific action-processing rule

```
(this is the investigate player's awareness before action rule):
consider the player's action awareness rules;
if rule succeeded, change within the player's sight to true;
otherwise change within the player's sight to false.
```

A specific action-processing rule (this is the check stage rule):

```
anonymously abide by the specific check rulebook.
```

A specific action-processing rule (this is the carry out stage rule):

```
consider the specific carry out rulebook.
```

A specific action-processing rule (this is the after stage rule):

```
if action in world is true, abide by the after rules.
```

A specific action-processing rule

```
(this is the investigate player's awareness after action rule):
if within the player's sight is false:
    consider the player's action awareness rules;
    if rule succeeded, change within the player's sight to true;
```

A specific action-processing rule (this is the report stage rule):

```
if within the player's sight is true and action keeping silent is false,
    consider the specific report rulebook;
```

The last specific action-processing rule: rule succeeds.

§39. The unusual use of “anonymously abide by” above is a form of “abide by” which may be worth explaining. Suppose rule X consists of an instruction to abide by rulebook Y , and suppose that Y in fact fails when it reaches Y_k . If the ordinary “abide by” is used then the action will be deemed to have failed at rule X , but if “anonymously abide by” is used then it will be deemed to have failed at Y_k . (Thus X remains anonymous: it can never be the culprit.) We only use this at the check stage, because the carry out, report and after stages are not allowed to fail the action. (The after stage is allowed to end it, but not in failure.)

§40. The specific action-processing rulebook is probably more fruitful than the main one if we want to modify what happens. For instance:

This is the sixth sense rule: if the player is not the actor, say "You sense that [the actor] is up to something."

The sixth sense rule is listed before the carry out stage rule in the specific action-processing rules.

...produces the message at a time when the action is definitely possible and will succeed, but before anything has been done.

§41. The only rule not spelled out was the primitive “work out details of specific action rule”, which initialises the rulebook variables so that they record the action’s specific check, carry out and report rulebooks, and whether or not it is in world.

The work out details of specific action rule translates into I6 as "WORK_OUT_DETAILS_OF_SPECIFIC_R".

§42. Player's action awareness. This rulebook decides whether or not an action by somebody should be routinely reported to the player: is he aware of it having taken place? If the rulebook positively succeeds then he is, and otherwise not.

A player's action awareness rule

(this is the player aware of his own actions rule):

if the player is the actor, rule succeeds.

A player's action awareness rule

(this is the player aware of actions by visible actors rule):

if the player is not the actor and the player can see the actor, rule succeeds.

A player's action awareness rule

(this is the player aware of actions on visible nouns rule):

if the noun is a thing and the player can see the noun, rule succeeds.

A player's action awareness rule

(this is the player aware of actions on visible second nouns rule):

if the second noun is a thing and the player can see the second noun, rule succeeds.

§43. Accessibility. The "accessibility" rulebook is not very visible to users: it's another behind-the-scenes rulebook for managing the decision as to whether the actor can touch any items which the intended action requires him to be able to reach.

In its default configuration, it contains only the "access through barriers" rule. This in all circumstances either succeeds or fails: in other words, it makes a definite decision, and this is why it is anchored as "last" in the rulebook. If users or extensions want to tweak accessibility at this general level, any new rules they add will get the chance to decide before the "access through barriers" rule settles the matter. But in practice we expect most users to work with one of the two reaching rulebooks instead.

Section SR2/11 - Accessibility

The access through barriers rule is listed last in the accessibility rules.

The access through barriers rule translates into I6 as "ACCESS_THROUGH_BARRIERS_R".

§44. Reaching inside. What the access through barriers rule does is to try to construct a path through the object containment tree from the actor to the item needed, and to apply the reaching inside or reaching outside rulebooks each time this path passes inside or outside of a container (or room). (Supporters never form barriers.)

The can't reach inside rooms rule is listed last in the reaching inside rules. [Penultimate.]

The can't reach inside closed containers rule is listed last in the reaching inside rules. [Last.]

The can't reach inside closed containers rule translates into I6 as "CANT_REACH_INSIDE_CLOSED_R".

The can't reach inside rooms rule translates into I6 as "CANT_REACH_INSIDE_ROOMS_R".

§45. Reaching outside. And, not quite symmetrically since we don't need to block room-to-room reaching on both the inbound and outbound directions,

The can't reach outside closed containers rule is listed last in the reaching outside rules.

The can't reach outside closed containers rule translates into I6 as "CANT_REACH_OUTSIDE_CLOSED_R".

§46. Visibility. The visibility rulebook actually has the opposite sense to the one the name might suggest. It is applied only when the player (not any other actor) tries to perform an action which requires light to see by; the action is blocked if the rulebook succeeds. (Well, but we could hardly call it the invisibility rulebook. The name is supposed to suggest that the consideration of visibility is being applied: rather the way cricket matches end with a declaration that “light stopped play”, meaning of course that darkness did.)

The can't act in the dark rule is listed last in the visibility rules.

The last visibility rule (this is the can't act in the dark rule): if in darkness, rule succeeds.

§47. Does the player mean. This rulebook is akin to a set of preferences about how to interpret the player's commands in cases of ambiguity. No two Inform users are likely to agree about the best way to decide these, so we are fairly hands-off, and make only one rule as standard:

Does the player mean taking something which is carried by the player
 (this is the very unlikely to mean taking what's already carried rule):
 it is very unlikely.

§48. And that completes the creation and stocking of the 25 rulebooks. More than half of them are initially empty, including before, instead and after – at the end of the day, these rulebooks are hooks allowing the user to change the ordinary behaviour of things, but ordinariness is exactly what the Standard Rules is all about.

§49. Adjectives applied to values. There is a small stock of built-in adjectives for values.

Definition: a number is even rather than odd if the remainder after dividing it by 2 is 0.

Definition: a number is positive if it is greater than zero.

Definition: a number is negative if it is less than zero.

Definition: a text is empty rather than non-empty if it is "".

Definition: an indexed text is empty rather than non-empty if I6 routine
 "INDEXED_TEXT_TY_Empty" says so (it contains no characters).

A scene can be recurring or non-recurring. A scene is usually non-recurring.

Definition: a scene is happening if I6 condition "scene_status-->(*1-1)==1"
 says so (it is currently taking place).

Definition: a table-name is empty rather than non-empty if the number of filled rows in it is 0.

Definition: a table-name is full rather than non-full if the number of blank rows in it is 0.

Definition: a rulebook is empty rather than non-empty if I6 routine "RulebookEmpty" says so (it contains no rules, so that following it does nothing and makes no decision).

Definition: an activity is empty rather than non-empty if I6 routine "ActivityEmpty" says so (its before, for and after rulebooks are all empty).

Definition: an activity is going on if I6 routine "TestActivity" says so (one of its three rulebooks is currently being run).

Definition: a list of values is empty rather than non-empty if I6 routine
 "LIST_OF_TY_Empty" says so (it contains no entries).