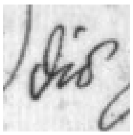
Chapter V: Natural Language

Westlich von Haus
Du stehst auf freiem Feld westlich von einem weißen Haus, dessen Haustür mit Brettern vernagelt ist. Hier ist ein kleiner Briefkasten.
— 'Zork I: Das Große Unterweltreich'

§34 Linguistics and the Inform parser



¡Bienvenido a Aventura! Despite the English-language bias of early computers and their manuals, interactive fiction has a culture and a history beyond English, not least in Germany. Like the Monty Python team and the Beatles, Infocom made a German translation of their defining work, when in early 1988 Jeff O'Neill coded up 'Zork I: Das Große Unterweltreich'. It came at a sorry time in Infocom's fortunes and remains officially unreleased, in part because the translator recruited had rendered the text in a stilted, business-German manner, though a beta-test of the story file circulates to this day. But O'Neill's work was not in vain, as it left another important legacy: an upgrading of the Z-machine format to allow for accented characters, which opened the door to non-English IF on the Z-machine. Jose Luiz Diaz's translation of 'Advent' into Spanish, as 'Aventura', stimulated much of the 1996 development of Inform as a multilingual system, and Toni Arnold's game 'Rummelplatzgeschichte' (1998) also deserves mention, as does advice from Inform users across four continents, among them Torbjörn Andersson, Joachim Baumann, Paul David Doherty, Bjorn Gustavsson, Aapo Haapanen, Ralf Herrmann, J. P. Ikaheimonen, Ilario Nardinocchi, Bob Newell, Giovanni Riccardi and Linards Ticmanis. If nothing else, I am glad to have learned the palindromic Finnish word for soap dealer, "saippuakauppias".

The standard English-language release of the Inform library now consists of eight files of code. Of these eight, only two need to be replaced to make a translation to another language: `"Grammar.h"`, which contains grammars for English verbs like "take" and "drop"; and a "language definition file" called `"English.h"`. For instance, in Ilario Nardinocchi's translation these two files are replaced by `"ItalianG.h"` and `"Italian.h"`, in Jose Luis Diaz's translation they become `"SpanishG.h"` and `"Spanish.h"` and so on. Language definition files can be useful for more, or rather less, than just translation. 'The Tempest' (1997), for instance, uses a language definition file to enable it to speak in Early Modern English verse and to recognise pronouns like "thee" and "thy". A suitable language definition file could also change the persona of an Inform game from second-person ("You fall into a pit!") to first-person ("I have fallen into a pit!") or third ("Bilbo falls into a pit!"), or from present to past tenses, as Jamie Murphy's game 'Leopold the Minstrel' (1996) did.

This section goes into the linguistics of the Inform parser, and how to add new grammatical concepts to it using grammar tokens. The next goes into full-scale translation and how to write new language definition files.

. . . . .

Language is complex, computers are simple. Modern English is a mostly non-inflected language, meaning that words tend not to alter their spelling according to their usage, but even here the parser has to go to some trouble to cope with one of its remaining inflections ("take coin" but "take six coins": see §29). The range of variation in human languages is large and as many are heavily inflected the task at first seems hopeless.†

On the other hand, Inform is mainly used with Romance-family languages, where commands are formed roughly as they are in English. The language understood by the parser is a simple one, called Informese. It has three genders, two numbers, a concept of animate versus inanimate nouns and a clear understanding of articles and pronouns, but all verbs are imperative, the only tense is the present, there are no cases of nouns (but see §35) and adjectives are not distinguished from nouns (but see §26). Informese is based on a small part of English, but the proposition of this chapter is that (with some effort) you can find Informese at the core of many other languages as well.

Changes of vocabulary are obviously needed: for instance, where an English game recognises "other", a French one must recognise "autre". But, as the following example shows, vocabulary changes are not enough:

jetez la boule dedans	*throw the ball into it* (French)

has no word-for-word translation into Informese, because "dedans" (into it) is a pronominal adverb, and Informese doesn't have pronominal adverbs.

† In fact the difficult languages to parse are not those with subtleties of spelling but those where even word-recognition can be a matter of context and guesswork, such as Hebrew, where all vowels are conventionally omitted.

Instead, a transformational rule like this one must be applied:

dedans *inside it* ↦ dans lui

Transformational rules like this one bring new grammatical structures into the Inform parser. The rest of this section is occupied with describing what is present already.

· · · · ·

The following is a short grammar of Informese. Both here and in the General Index, grammatical concepts understood by the parser are written in angle brackets ⟨like so⟩.

(1) *Commands*

A command to an Inform game should be one of:

⟨oops-word⟩ ⟨word⟩
⟨action phrase⟩
⟨noun phrase⟩, ⟨action phrase⟩

An ⟨oops-word⟩ corrects the last command by putting the ⟨word⟩ in to replace whatever seemed to be incorrect. In `"English.h"`, the only words in the ⟨oops-word⟩ category are "oops" and its abbreviation "o". An ⟨action phrase⟩ instructs the player to perform an action, unless it is preceded by a ⟨noun phrase⟩ and a comma, in which case someone else is instructed to perform an action.

An ⟨action phrase⟩ consists of a sequence of verb phrases, divided up by full stops or then-words: a ⟨then-word⟩ is a word like the English "then" or a full stop. For instance "take sword. east. put sword in stone" is broken into a sequence of three verb phrases, each parsed and acted on in turn. (It's important not to parse them all at once: the meaning of the noun phrase "stone" depends on where the player is by then.)

(2) *Verb phrases*

A ⟨verb phrase⟩ is one of:

⟨again-word⟩
⟨imperative verb⟩ ⟨grammar line⟩

Again-words are another category: in `"English.h"` these are "again" and its abbreviation "g". An ⟨again-word⟩ is understood as "the ⟨verb phrase⟩ most recently typed in which wasn't an ⟨again-word⟩".

The imperative is the form of the verb used for orders or instructions. In English the imperative ("open the window") looks the same as the infinitive ("to open"), but in most languages they differ (French "ouvrez" is imperative, "ouvrir" infinitive). Even in many languages where verbs usually follow objects, such as Latin, the imperative comes at the start of a verb phrase, and Informese insists on this. Informese also wants the ⟨imperative verb⟩ to be a single word, but programming can get around both requirements.

Grammar lines are sequences of tokens. Each token results in one of four grammatically different outcomes:

⟨noun phrase⟩
⟨preposition⟩
⟨number⟩
⟨unparsed text⟩

For instance, a successful match for the tokens `noun` or `multiheld` would produce a ⟨noun phrase⟩, whereas a match for `'into'` would produce a ⟨preposition⟩. Note that a general parsing routine can produce any of these four outcomes.

(3) *Prepositions*

Any word written in quotes as a grammar token. This is normally also a preposition in the ordinary grammatical sense, but not always, as the "press charges" example in §30 shows. In `"English.h"`, "look under table" and "switch on radio" contain two words considered to be prepositions in Informese: "under" and "on".

(4) *Numbers*

Include at least the numbers 1 to 20 written out in words. "At least" because a language definition file is free to include more, but should not include less.

(5) *Noun phrases*

A string of words which refer to a single object or collection of objects, with more or less exactness. Here are some typical examples of `"English.h"` noun phrases:

it
rucksack
brown bag, pepper
a box and the other compass
nine silver coins
everything except the rucksack
smooth stones

A noun phrase is a list of basic noun phrases:

⟨basic np⟩ ⟨connective⟩ ⟨basic np⟩ ⟨connective⟩ . . . ⟨connective⟩ ⟨basic np⟩

and there are two kinds of connective: an ⟨and-word⟩ (conjunction), and a ⟨but-word⟩ (disjunction). The Inform parser always regards a comma in a ⟨noun phrase⟩ (other than one used at the start of a command: see (1) above) as an ⟨and-word⟩, and the definition of `"English.h"` gives "and" as another. `"English.h"` has two ⟨but-words⟩: "but" and "except".

⟨Noun phrases⟩ being parsed are assigned several properties. They are declared *definite* if they carry no article, or a definite article which is not qualified by an all-word or a demanding number, and are otherwise *indefinite*. (Except that a noun-phrase containing a dictionary word flagged as likely to be referring to plural objects, such as `'crowns//p'`, is always indefinite.) Definiteness affects disambiguation and the parser's willingness to make guesses, as the description of the parser's disambiguation algorithm at the end of §33 shows.

Indefinite noun phrases also have a target quantity of objects being referred to: this is normally 1, but 7 for "seven stones" and 100, used internally to mean "as many as possible", for "crowns" or "all the swords". Noun phrases also have a *gender-number-animation* combination, or "GNA":

Gender: in most European languages, nouns divide up into masculine, feminine or neuter, the three genders in Informese. Gender is important when parsing noun phrases because it can distinguish otherwise identical nouns, as in French: "le faux", the forgery, "la faux", the scythe. As in German, there may be no satisfactory way to determine the gender of a noun by any automatic rules: see the next section for how Inform assigns genders to nouns.

Number: singular ("the hat") or plural ("the grapes"). Individual objects in Inform games can have names of either number. Languages with more than two numbers are rare, but Hebrew has a "pair of" number. This would have to be translated into a demanding number (see (7d) below) for Informese.

Animation: Informese distinguishes between the animate (people and higher animals) and the inanimate (objects, plants and lower animals).

With three genders, two numbers and two animations, Informese has twelve possible GNA combinations, and these are internally represented by the numbers 0 to 11:

0	animate	singular	masculine
1			feminine
2			neuter
3		plural	masculine
4			feminine
5			neuter
6	inanimate	singular	masculine
7			feminine
8			neuter
9		plural	masculine
10			feminine
11			neuter

Not all possible GNAs occur in all natural languages. In English, cases 6, 7, 9 and 10 never occur, except perhaps that ships are sometimes called "she" and "her" without being animate (GNA 7). In French, 2, 5, 8 and 11 never occur. The parser actually works by assigning sets of possible GNA values to each noun phrase: so, in French, "le faux" carries the set $\{6\}$, while the more ambiguous noun phrase "les" carries $\{3, 4, 9, 10\}$.

(6) *Basic noun phrases*

These take the following form, in which both lists can have any number of words in, including none, and in any order:

⟨list of descriptors⟩ ⟨list of nouns⟩

For instance "the balloon" has one descriptor and one noun; "red balloon" has just two nouns; "all" has just one descriptor.

(7) *Descriptors*

There are five kinds of ⟨descriptor⟩, as follows:

(a) An ⟨article⟩ is a word indicating whether a particular object is being referred to, or merely one of a range. Thus there are two kinds of article, *definite* and *indefinite*. `"English.h"` has four articles: "the" is definite, while "a", "an" and "some" are indefinite.

(b) An ⟨all-word⟩ is a word which behaves like the English word "all", that is, which refers to a whole range of objects. Informese, like some natural languages (such as Tagalog), handles this as a kind of article but which pluralises what follows it.

(c) An ⟨other-word⟩ is a word behaving like "other", which Informese understands as "other than the one I am holding". Thus, if the player is holding a sword in a room where there's also a sword on the floor, then "examine other sword" would refer to the one on the floor.

(d) A ⟨demanding number⟩ is a word like "nine" in "nine bronze coins", which demands that a certain number of items are needed.

(e) A ⟨possessive adjective⟩ is a word indicating ownership by someone or something whose meaning is held in a pronoun. Among others `"English.h"` has "my" (belonging to "me"); French has "son" (belonging to "lui"). Informese also counts ⟨demonstrative adjectives⟩ like "this", "these", "that" and "those" as possessives, though demonstratives are hardly ever used by players and may not be worth providing in other languages. In Spanish, for instance, there would have to be twelve, for "this", "that" (nearby) and "that" (far away), each in masculine, feminine, singular and plural forms; and the structure of "celui-ci" and "celui-la" in French is too complex to be worth the effort of parsing.

(8) *Nouns*

There are three kinds of ⟨noun⟩, as follows:

(a) A ⟨name⟩ is a word matched against particular objects. Unless an object has a `parse_name` routine attached, which complicates matters, these will be the words in its `name` array. For instance:

```
Object -> "blue box" with name 'blue' 'box';
```

has two possible names in Informese, "blue" and "box".

(b) A ⟨me-word⟩ is a word which behaves like the English word "me", that is, which refers to the player-object. Most languages would think these are just examples of relative pronouns, but Informese considers them to be in a category of their own. Note that they refer to the player, whoever is addressed: in "mark, give me the bomb", "me" refers to the speaker, not to Mark.

(c) A ⟨pronoun⟩ is a word which stands in the place of nouns and can only be understood with reference back to what has previously been said. "it", "him", "her" and "them" are all pronouns in `"English.h"`. (Though "her" can also be a possessive adjective, as in (7e) above.)

· · · · ·

It is worth mentioning a number of grammatical features which are *not* contained in Informese, along with some ways to simulate them.

adverbs such as "quickly" in "run quickly east". These are not difficult to implement:

```
Verb 'run'
    * noun=ADirection -> Go
    * 'quickly' noun=ADirection -> GoQuickly
    * noun=ADirection 'quickly' -> GoQuickly;
```

However, "The authors of Zork have thought about several possible extensions to the Zork parser. One that has come up many times is to add adverbs. A player should be able to do the following:

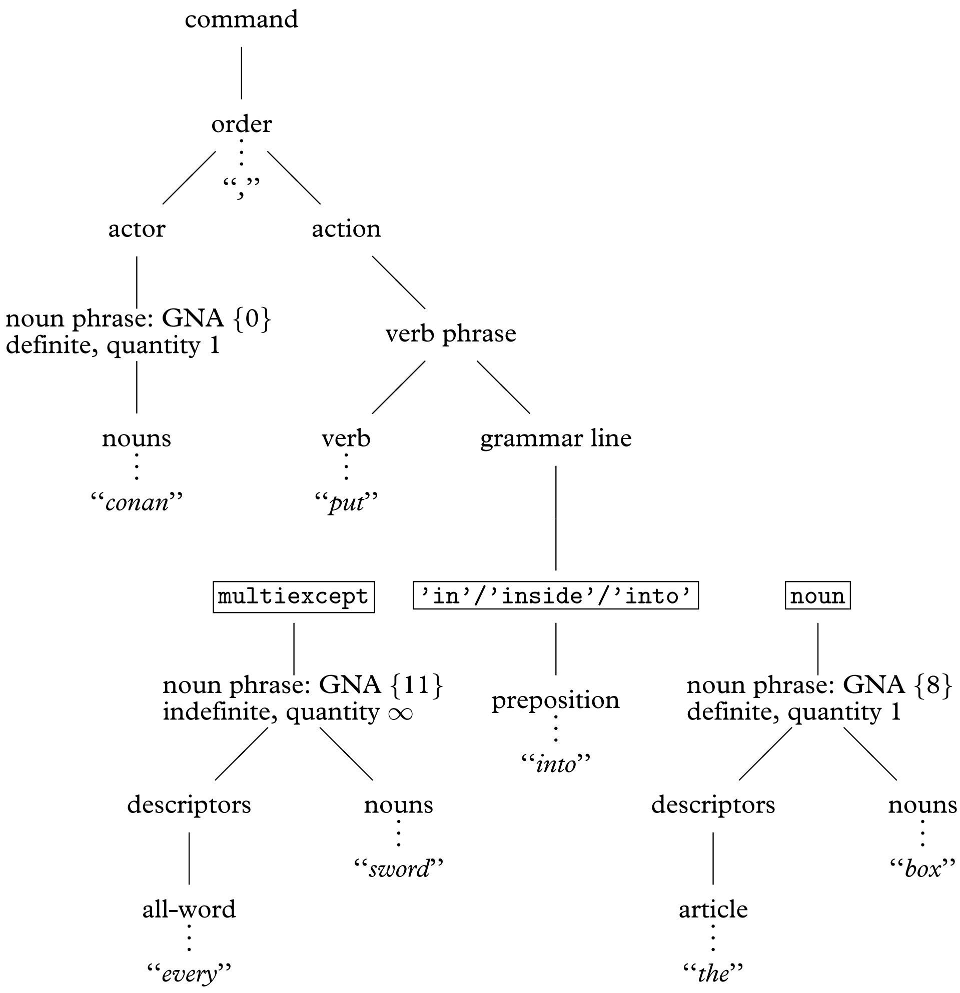
```
>go north quietly
```
You sneak past a sleeping lion who sniffs but doesn't wake up.

The problem is to think of reasons why you would not do everything 'quietly', 'carefully' or whatever." (P. David Lebling, "Zork and the Future of Computerized Fantasy Simulations", *Byte*, December 1980.) A further problem is the impracticality of modelling the game world closely enough to differentiate between ways to achieve the same action. In Melbourne House's 'The Hobbit' adverbs influence the probability of success in randomised events, so for instance "throw rope vigorously across river" is more likely to succeed than "throw rope across river", but those few players who discovered this were not pleased. Twenty years on from 'Zork', adverbs remain largely unused in the medium.

adjectives are not distinguished from nouns, although it can be useful to do so when resolving ambiguities. See §28 for remedies.

genitives: objects are not normally named by description of their circumstances, so "the box on the floor" and "the priest's hat" would not normally be understood. Designers can still define objects like

```
Object -> "priest's hat"
    with name 'hat' 'priest^s';
```



An example of parsing Informese. This diagram shows the result of parsing the text "conan, put every sword into the box", assuming that the verb "put" has a grammar line reading

```
* multiexcept 'in'/'inside'/'into' noun -> Insert
```

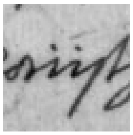
as indeed it does have in the `"English.h"` grammar.

in which the genitive "priest's" is a noun like any other.

pronouns of other kinds, notably: nominative pronouns ("I" in "I am happy"); interrogative pronouns ("What" in "What are you doing?"), although these are often simulated by making "what" an Informese verb; demonstrative pronouns ("that" in "eat that"), although in `"English.h"` the parser gets this right because they look the same as demonstrative adjectives with no noun attached; possessive pronouns ("mine" in "take the troll's sword. give him mine", which should expand "mine" to "my *X*", where *X* is the current value of "it").

pronominal adverbs are not found in English, but are common in other languages: for instance "dessous" (French: "under it"). The next section suggests how these can be achieved.

§35 Case and parsing noun phrases



As this section and the next use a variety of linguistic terms, here are some definitions. "Flexion" is the changing of a word according to its situation, and there are several kinds:

inflection: a variable ending for a word, such as "a" becoming "an".

agreement: when the inflection of one word is changed to match another word which it goes with. Thus "grand maison" but "grande dame" (French), where the inflection on "grand" agrees with the gender of the noun it is being applied to.

affix: part of a word which is attached either at the beginning (*prefix*), the end (*suffix*) or somewhere in the middle (*infix*) of the ordinary word (the *stem*) to indicate, for instance, person or gender of the objects attached to a verb. The affix often plays a part that an entirely separate word would play in English. For instance, "donnez-lui" (French: "give to him"), where the suffix is "-lui", or "cogela" (Spanish: "take it"), where the suffix is "la".

enclitic: an affix, usually a suffix, meaning "too" or "and" in English. For instance, "que" (Latin).

agglutinization: the practice of composing many affixes to a single word, so that it may even become an entire phrase. For instance:

kirjoitettuasi	*after you had written* (Finnish)

. . . . .

In most languages, noun phrases have different *cases* according to their situation in a sentence. In the English sentence "Emily ate one bath bun and gave Beatrice the other", the noun phrase "Emily" is *nominative*, "one bath bun" and "the other" are *accusative* and "Beatrice" is *dative*. These last two are the cases most often occurring in Inform commands, as in the example

leg den frosch auf ihn	*put the frog on him* (German)
nimm den frosch von ihm	*take the frog from him*

where the noun phrase "den frosch" is accusative both times, but "ihn" and "ihm" are the same word ("him") in its accusative and dative forms. In some languages a *vocative* case would also be needed for the name of someone being addressed:

domine, fiat lux	*Lord, let there be light* (Latin)

since “domine” is the vocative form of “dominus”. Latin also has genitive and ablative cases, making six in all, but this pales by comparison with Finnish, which has about thirty. In effect, a wide range of English prepositional phrases like “into the water” are written as just the noun phrase “water” with a suffix meaning “into”.

. . . . .

To parse any of these languages, and even in some circumstances to parse special effects in English-language games, it’s useful to have further control over the way that the parser recognises noun phrases.

The words entered into an object’s `name` property normally take the accusative case, the one most often needed in commands, as for example in the grammar line:

```
Verb 'take' * noun -> Take;
```

On the other hand, the nouns in the following grammar lines aren’t all accusative:

```
Verb 'give'
    * noun 'to' noun -> Give
    * noun noun      -> Give reverse;
```

This matches “give biscuit to jemima” and “give jemima biscuit”, “biscuit” being accusative in both cases and “to jemima” and “jemima” both dative. In a language where the spelling of a word can tell a dative from an accusative, such as German, we could instead use grammar like this:

```
Verb 'give'
    * noun dativenoun -> Give
    * dativenoun noun -> Give reverse;
```

where $\boxed{\texttt{dativenoun}}$ is some token meaning “like $\boxed{\texttt{noun}}$, but in the dative case instead of the accusative”. This could be used as the definition of a German verb “gib”, in which case both of the following would be parsed correctly:

gib die blumen dem maedchen	*give the flowers to the girl*
gib dem maedchen die blumen	*give the girl the flowers*

Unfortunately Inform doesn’t come with a token called $\boxed{\texttt{dativenoun}}$ built in, so you have to write one, using a general parsing routine (see §31). For the sake of an example closer to English, suppose a puzzle in which the Anglo-Saxon hero Beowulf will do as he is asked to, but only if addressed in Old English:

beowulf, gief gold to cyning	*beowulf, give gold to king* (Old English)

The grammar would be much like that for German, and indeed English:

```
Verb 'gief' * noun dativenoun -> OEGive;
```

and here is a simple version of `dativenoun`:

```
[ dativenoun;
  if (NextWord() == 'to')
      return ParseToken(ELEMENTARY_TT, NOUN_TOKEN);
  return GPR_FAIL;
];
```

Read this as: "if the next word is "to", try and match a noun following it; otherwise it isn't a dative". A more likely form of the command is however

beowulf, gief gold cyninge	*beowulf, give gold to king* (Old English)

where "cyninge" is the dative form of "cyning". The ending "-e" often indicates a dative in Old English, but there are irregularities, such as "searo" (device), whose dative is "searwe", not "searoe". In the unlikely event of Beowulf confronting a vending machine:

beowulf, gief gold to searo	*beowulf, give gold to device* (Old English)
beowulf, gief gold searwe	*beowulf, give gold to device*

How to manage all this? Here is a laborious way:

```
Object -> "searo"
  with name 'searo', dativename 'searwe';
Object -> "Cyning"
  with name 'cyning', dativename 'cyninge';
[ dativenoun;
  if (NextWord() ~= 'to') {
      wn--; parser_inflection = dativename;
  }
  return ParseToken(ELEMENTARY_TT, NOUN_TOKEN);
];
```

The variable `parser_inflection` tells the parser where to find the name(s) of an object. It must always be equal to *either* a property *or* a routine. Most of the time it's equal to the property `name`, the accusative case as normal. If it equals another property, such as `dativename`, then the parser looks in that property for name-words instead of in `name`.

The above solution is laborious because it makes the game designer write out dative forms of every name, even though they are very often the same but with "-e" suffixed. It's for this kind of contingency that `parser_inflection` can be set to a routine name. Such an "inflection routine" is called with two arguments: an object and a dictionary word. It returns `true` if the dictionary word can mean the object and `false` if not. The word number `wn` is always set to the number of the next word along, and it should not be moved. Two library routines may be particularly helpful:

```
DictionaryLookup(text, length)
```

returns 0 if the word held as a `->` array of characters

```
text->0, text->1, ..., text->(length-1)
```

is not in the game's dictionary, or its dictionary entry if it is.

```
WordInProperty(word, object, property)
```

to see if this is one of the words listed in `object.property`. It may also be useful to know that the variable `indef_mode` is always set to `true` when parsing something known to be indefinite (e.g., because an indefinite article or a word like "all" has just been typed), and `false` otherwise.

• △**EXERCISE 107**
Rewrite the `dativenoun` token so that "-e" is recognised as a regular suffix indicating the dative, while still making provision for some nouns to have irregular dative forms.

• △**EXERCISE 108**
Now add an (imaginary, not Old English) dative pronominal adverb "toit", which is to be understood as "to it".

• △△**EXERCISE 109**
Define a token `swedishnoun` to make nouns and adjectives agree with the article (definite or indefinite) applied to them, so for instance:

en brun hund	*a brown dog* (Swedish)
den bruna hunden	*the brown dog*
ett brunt hus	*a brown house*
det bruna huset	*the brown house*

△△ The use of grammar tokens is only one way of dealing with flexion and pronominal adverbs. The alternative is to alter the text typed until it resembles normal Informese:

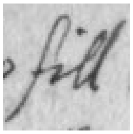
gief gold cyninge $\mapsto$ gief gold to cyning
gief gold toit $\mapsto$ gief gold to it
den bruna hunden $\mapsto$ den brun hund
det bruna huset $\mapsto$ det brun hus

See §36 below. In a heavily inflected language with many irregularities, a combination of the two techniques may be needed.

§36 Parsing non-English languages

It, hell. She had *Those*.

— Dorothy Parker (1893–1967), reviewing the romantic novel '*It*'



Before embarking on a new language definition file, the translator may want to consider what compromises are worth making, omitting tricky but not really necessary features of the language. For instance, in German, adjectives take forms agreeing with whether their noun takes a definite or indefinite article:

ein großer Mann	*a tall man* (German)
der große Mann	*the tall man*

This is an essential. But German also has a "neutral" form for adjectives, used in sentences like

Der Mann ist groß *The man is tall*

Now it could be argued that if the parser asks the German equivalent of

Whom do you mean, the tall man or the short man?

then the player ought to be able to reply "groß". But this is probably not worth the effort.

As another example from German, is it essential for the parser to recognise commands put in the polite form when addressed to somebody other than the player? For instance,

freddy, öffne den ofen	*Freddy, open the oven*
herr krüger, öffnen sie den ofen	*Mr Krueger, open the oven*

indicate that Freddy is a friend but Mr Krueger a mere acquaintance. A translator might go to the trouble of implementing this, but equally might not bother, and simply tell players always to use the familiar form. It's harder to avoid the issue of whether the computer is familiar to the player. Does the player address the computer or the main character of the game? In English it makes no difference, but there are languages where an imperative verb agrees with the gender of the person being addressed. Is the computer male? Is it still male if the game's main character is female?

Another choice is whether to require the player to use letters other than 'a' to 'z' from the keyboard. French readers are used to seeing words written in capital letters without accents, so that there is no need to make the player type accents. In Finnish, though, 'ä' and 'ö' are significantly different from 'a' and 'o': "vaara" means "danger", but "väärä" means "wrong".

Finally, there are also dialect forms. The number 80 is "quatre-vingt" in metropolitan French, "octante" in Belgian and "huitante" in Swiss French. In such cases, the translator may want to write the language definition file to cope with all possible dialects. For example, something like

```
#ifdef DIALECT_FRANCOPHONE; print "septante";
#ifnot; print "soixante-dix";
#endif;
```

would enable the same definition file to be used by Belgian authors and members of the Académie française alike. The standard "English.h" definition already has such a constant: DIALECT_US, which uses American spellings, so that if an Inform game defines

```
Constant DIALECT_US;
```

before including Parser, then (for example) the number 106 would be printed in words as "one hundred six" instead of "one hundred and six".

△ An alternative is to allow the player to change dialect during play, and to encode all spelling variations inside variable strings. Ralf Herrmann's "German.h" does this to allow the player to choose traditional, reformed or Swiss German conventions on the use of "ß". The low string variables @30 and @31 each hold either "ss" or "ß" for use in words like "schlie@30t" and "mu@31t".

· · · · ·

Organisation of language definition files

A language definition file is itself written in Inform, and fairly readable Inform at that: you may want to have a copy of "English.h" to refer to while reading the rest of this section. This is divided into four parts:

I. Preliminaries
II. Vocabulary
III. Translating to Informese
IV. Printing

It is helpful for all language definitions to follow the order and layout style of "English.h". The example used throughout the rest of the section is of developing a definition of "French.h".

(I.1) Version number and alphabet

The file should begin as follows:

```
! ==============================================================
!   Inform Library Definition File: French
!
!   (c) Angela M. Horns 1996
! --------------------------------------------------------------
System_file;
! --------------------------------------------------------------
!   Part I.   Preliminaries
! --------------------------------------------------------------
Constant LanguageVersion
    = "Traduction fran@ccais 961205 par Angela M. Horns";
```

("English.h" defines a constant called EnglishNaturalLanguage here, but this is just to help the library keep old code working with the new parser: don't define a similar constant yourself.) Note the c-cedilla written using escape characters, @cc not c, which is a precaution to make absolutely certain that the file will be legible on anyone's system, even one whose character set doesn't have accented characters in the normal way.

The next ingredient of Part I is declaring the accented letters which need to be "cheap" in the following sense. Inside story files, dictionary words are stored to a "resolution" of nine Z-characters: that is, only the first nine Z-characters are looked at, so that

"chrysanthemum"	is stored as	'chrysanth'
"chrysanthemums"	is stored as	'chrysanth'

(This is one of the reasons why Informese doesn't make linguistic use of word-endings.) Normally no problem, but unfortunately Z-characters are not the same as letters. The letters 'A' to 'Z' are "cheap" and take only one Z-character each, but accented letters like 'é' normally take out four Z-characters. If your translation is going to ask the player to type accented letters at the keyboard (which even a French translation need not do: see above), the resolution may be unacceptably low:

"télécarte"	is stored as	't@'el'
"téléphone"	is stored as	't@'el'

as there are not even enough of the nine Z-characters left to encode the second 'é', let alone the 'c' or the 'p' which would distinguish the two words. Inform therefore provides a mechanism to make up to about 10 common accents cheaper to use, in that they then take only two Z-characters each, not four. In the case of French, we might write:

```
Zcharacter '@'e';    ! E-acute
Zcharacter '@`e';    ! E-grave
Zcharacter '@`a';    ! A-grave
Zcharacter '@`u';    ! U-grave
Zcharacter '@^a';    ! A-circumflex
Zcharacter '@^e';    ! E-circumflex
```

(Note that since the Z-machine automatically reduces anything the player types into lower case, we need only include lower-case accented letters here. Note also that there are plenty of other French accented letters (ï, û and so forth) but these are uncommon enough not to matter here.) With this change made,

"télécarte"	is stored as	`'t@'el@'ecar'`
"téléphone"	is stored as	`'t@'el@'epho'`

enabling a phone card and a phone to be correctly distinguished by the parser.

△△ In any story file, 78 of the characters in the ZSCII set are designated as "cheap" by being placed into what's called the "alphabet table". One of these is mandatorily new-line, another is mandatorily double-quote and a third cannot be used, leaving 75. `Zcharacter` moves a ZSCII character into the alphabet table, throwing out a character which hasn't yet been used to make way. Alternatively, and provided no characters have so far been used at all, you can write a `Zcharacter` directive which sets the entire alphabet table. The form required is to give three strings containing 26, 26 and 23 ZSCII characters respectively. For instance:

```
Zcharacter "abcdefghijklmnopqrstuvwxyz"
           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
           "0123456789!$&*():;.,<>@{386}";
```

Characters in alphabet 1, the top row, take only one Z-character to print; characters in alphabets 2 and 3 take two Z-characters to print; characters not in the table take four. Note that this assumes that Unicode $0386 (Greek capital Alpha with tonos accent, as it happens) is present in ZSCII. Ordinarily it would not be, but the block of ZSCII character codes between 155 and 251 is configurable and can in principle contain any Unicode characters of your choice. By default, if Inform reads ISO 8859-`n` (switch setting `-Cn`) then this block is set up to contain all the non-ASCII letter characters in ISO 8859-`n`. In the most common case, `-C1` for ISO Latin-1, the ligatures 'œ' and 'Œ' are then added, but this still leaves 28 character codes vacant.

```
Zcharacter table + '@{9a}';
```

adds Unicode character $009a, a copyright symbol, to ZSCII. Alternatively, you can instruct Inform to throw away all non-standard ZSCII characters and replace them with a fresh stock. The effect of:

```
    Zcharacter table '@{9a}' '@{386}' '@^a';
```

is that ZSCII 155 will be a copyright symbol, 156 will be a Greek capital alpha with tonos, 157 will be an a-circumflex and 158 to 251 will be undefined; and all other accented letters will be unavailable. Such `Zcharacter` directives must be made before the characters in question are first used in game text. You don't need to know the ZSCII values, anyway: you can always write `@{9a}` when you want a copyright symbol.

(I.2) Compass objects

All that is left in Part I is to declare standard compass directions. The corresponding part of `"English.h"`, given below, should be imitated as closely as possible:

```
Class CompassDirection
 with article "the", number
  has scenery;
Object Compass "compass" has concealed;
Ifndef WITHOUT_DIRECTIONS;
CompassDirection -> n_obj "north wall"
                    with name 'n//' 'north' 'wall',      door_dir n_to;
CompassDirection -> s_obj "south wall"
                    with name 's//' 'south' 'wall',      door_dir s_to;
CompassDirection -> e_obj "east wall"
                    with name 'e//' 'east' 'wall',       door_dir e_to;
CompassDirection -> w_obj "west wall"
                    with name 'w//' 'west' 'wall',       door_dir w_to;
CompassDirection -> ne_obj "northeast wall"
                    with name 'ne' 'northeast' 'wall', door_dir ne_to;
CompassDirection -> nw_obj "northwest wall"
                    with name 'nw' 'northwest' 'wall', door_dir nw_to;
CompassDirection -> se_obj "southeast wall"
                    with name 'se' 'southeast' 'wall', door_dir se_to;
CompassDirection -> sw_obj "southwest wall"
                    with name 'sw' 'southwest' 'wall', door_dir sw_to;
CompassDirection -> u_obj "ceiling"
                    with name 'u//' 'up' 'ceiling',      door_dir u_to;
CompassDirection -> d_obj "floor"
                    with name 'd//' 'down' 'floor',      door_dir d_to;
Endif;
CompassDirection -> out_obj "outside"
                    with                               door_dir out_to;
```

```
CompassDirection -> in_obj "inside"
                    with                              door_dir in_to;
```

For example, "French.h" would contain:

```
Class  CompassDirection
  with article "le", number
  has  scenery;
Object Compass "compas" has concealed;
...
CompassDirection -> n_obj "mur nord"
                    with name 'n//' 'nord' 'mur',       door_dir n_to;
```

(II.1) Informese vocabulary: the small categories

This is where small grammatical categories like ⟨again-word⟩ are defined. The following constants must be defined:

AGAIN*__WD	words of type ⟨again-word⟩
UNDO*__WD	words of type ⟨undo-word⟩
OOPS*__WD	words of type ⟨oops-word⟩
THEN*__WD	words of type ⟨then-word⟩
AND*__WD	words of type ⟨and-word⟩
BUT*__WD	words of type ⟨but-word⟩
ALL*__WD	words of type ⟨all-word⟩
OTHER*__WD	words of type ⟨other-word⟩
ME*__WD	words of type ⟨me-word⟩
OF*__WD	words of type ⟨of-word⟩
YES*__WD	words of type ⟨yes-word⟩
NO*__WD	words of type ⟨no-word⟩

In each case * runs from 1 to 3, except for ALL*__WD where it runs 1 to 5 and OF*__WD where it runs 1 to 4. ⟨of-words⟩ have not been mentioned before: these are used in the sense of "three of the boxes", when parsing a reference to a given number of things. They are redundant in English because the player could have typed simply "three boxes", but Inform provides them anyway.

In French, we might begin with:

```
Constant AGAIN1__WD   = 'encore';
Constant AGAIN2__WD   = 'c//';
Constant AGAIN3__WD   = 'encore';
```

Here we can't actually think of a third synonymous word for "again", but we must define AGAIN3__WD all the same, and must not allow it to be zero. And so on, through to:

```
Constant YES1__WD     = 'o//';
Constant YES2__WD     = 'oui';
Constant YES3__WD     = 'oui';
```

⟨yes-words⟩ and ⟨no-words⟩ are used to parse the answers to yes-or-no questions (oui-ou-non questions in French, of course). It causes no difficulty that the word "o" is also an abbreviation for "ouest" because they are used in different contexts. On the other hand, ⟨oops-words⟩, ⟨again-words⟩ and ⟨undo-words⟩ should be different from any verb or compass direction name.

After the above, a few further words have to be defined as possible replies to the question asked when a game ends. Here the French example might be:

```
Constant AMUSING__WD    = 'amusant';
Constant FULLSCORE1__WD = 'grandscore';
Constant FULLSCORE2__WD = 'grand';
Constant QUIT1__WD      = 'a//';
Constant QUIT2__WD      = 'arret';
Constant RESTART__WD    = 'reprand';
Constant RESTORE__WD    = 'restitue';
```

(II.2) Informese vocabulary: pronouns

Part II continues with a table of pronouns, best explained by example. The following table defines the standard English accusative pronouns:

```
Array LanguagePronouns table
  !  word      possible GNAs:                  connected to:
  !             a     i
  !             s  p  s  p
  !             mfnmfnmfnmfn
     'it'     $$001000111000                 NULL
     'him'    $$100000100000                 NULL
     'her'    $$010000010000                 NULL
     'them'   $$000111000111                 NULL;
```

The "connected to" column should always be created with `NULL` entries. The pattern of `1` and `0` in the middle column indicates which types of ⟨noun phrase⟩ might be referred to with the given ⟨pronoun⟩. This is really a concise way of expressing a set of possible GNA values, saying for instance that "them" can match against noun phrases with any GNA in the set $\{3, 4, 5, 9, 10, 11\}$.

The accusative and dative pronouns in English are identical: for instance "her" in "give her the flowers" is dative and in "kiss her" is accusative. French is richer in pronoun forms:

donne-le-lui	*give it to him/her*
mange avec lui	*eat with him*

Here "-lui" and "lui" are grammatically quite different, with one implying masculinity where the other doesn't. The table needed is:

```
Array LanguagePronouns table
```

```
  !  word       possible GNAs:                   connected to:
  !             a     i
  !             s  p  s  p
  !             mfnmfnmfnmfn
     '-le'    $$100000100000                   NULL
     '-la'    $$010000010000                   NULL
     '-les'   $$000110000110                   NULL
     '-lui'   $$110000110000                   NULL
     '-leur'  $$000110000110                   NULL
     'lui'    $$100000100000                   NULL
     'elle'   $$010000010000                   NULL
     'eux'    $$000100000100                   NULL
     'elles'  $$000010000010                   NULL;
```

This table assumes that "-le" can be treated as a free-standing word in its own right, not as part of the word "donne-le-lui", and section (III.1) below will describe how to bring this about. Note that "-les" and "-leur" are treated as synonymous: Informese doesn't (ordinarily) care that dative and accusative are different.

Using the "pronouns" verb in any game will print out current values, which may be useful when debugging the above table. Here is the same game position, inside the building at the end of the road, in parallel English, German and Spanish text:

> *English: 'Advent'*
> At the moment, "it" means the small bottle, "him" is unset, "her" is unset and "them" is unset.
>
> *German: 'Abenteuer'*
> Im Augenblick, "er" heisst der Schlüsselbund, "sie" heisst die Flasche, "es" heisst das Essen, "ihn" heisst der Schlüsselbund, "ihm" heisst das Essen und "ihnen" ist nicht gesetzt.
>
> *Spanish: 'Aventura'*
> En este momento, "-lo" significa el grupo de llaves, "-los" no está definido, "-la" significa la pequeña botella, "-las" significa las par de tuberí as de unos 15 cm de diámetro, "-le" significa la pequeña botella, "-les" significa las par de tuberí as de unos 15 cm de diámetro, "él" significa el grupo de llaves, "ella" significa la pequeña botella, "ellos" no está definido y "ellas" significa las par de tuberí as de unos 15 cm de diámetro.

(II.3) Informese vocabulary: descriptors

Part II continues with a table of descriptors, in a similar format.

```
Array LanguageDescriptors table
  !  word       possible GNAs    descriptor      connected
  !             to follow:       type:           to:
  !             a     i
  !             s  p  s  p
  !           mfnmfnmfnmfn
     'my'     $$111111111111    POSSESS_PK      0
     'this'   $$111000111000    POSSESS_PK      0
     'these'  $$000111000111    POSSESS_PK      0
     'that'   $$111111111111    POSSESS_PK      1
     'those'  $$000111000111    POSSESS_PK      1
     'his'    $$111111111111    POSSESS_PK      'him'
     'her'    $$111111111111    POSSESS_PK      'her'
     'their'  $$111111111111    POSSESS_PK      'them'
     'its'    $$111111111111    POSSESS_PK      'it'
     'the'    $$111111111111    DEFART_PK       NULL
     'a//'    $$111000111000    INDEFART_PK     NULL
     'an'     $$111000111000    INDEFART_PK     NULL
     'some'   $$000111000111    INDEFART_PK     NULL;
```

This gives three of the four types of ⟨descriptor⟩. The constant `POSSESS_PK` signifies a ⟨possessive adjective⟩, connected either to 0, meaning the player-object, or to 1, meaning anything other than the player-object (used for "that" and similar words) or to the object referred to by the given ⟨pronoun⟩, which must be one of those in the pronoun table. `DEFART_PK` signifies a definite ⟨article⟩ and `INDEFART_PK` an indefinite ⟨article⟩: these should both give the connected-to value of `NULL` in all cases.

The fourth kind allows extra descriptors to be added which force the objects that follow to have, or not have, a given attribute. For example, the following three lines would implement "lit", "lighted" and "unlit" as adjectives automatically understood by the English parser:

```
    'lit'     $$111111111111    light           NULL
    'lighted' $$111111111111    light           NULL
    'unlit'   $$111111111111    (-light)        NULL
```

An attribute name means "must have this attribute", while the negation of it means "must not have this attribute".

To continue the example, `"French.h"` needs the following descriptors table:

```
Array LanguageDescriptors table
  !  word       possible GNAs   descriptor      connected
  !             to follow:      type:           to:
  !             a     i
  !             s  p  s  p
  !             mfnmfnmfnmfn
     'le'     $$100000100000    DEFART_PK       NULL
     'la'     $$010000010000    DEFART_PK       NULL
     'l^'     $$110000110000    DEFART_PK       NULL
     'les'    $$000110000110    DEFART_PK       NULL
     'un'     $$100000100000    INDEFART_PK     NULL
```

```
    'une'    $$010000010000    INDEFART_PK     NULL
    'des'    $$000110000110    INDEFART_PK     NULL
    'mon'    $$100000100000    POSSESS_PK      0
    'ma'     $$010000010000    POSSESS_PK      0
    'mes'    $$000110000110    POSSESS_PK      0
    'son'    $$100000100000    POSSESS_PK      '-lui'
    'sa'     $$010000010000    POSSESS_PK      '-lui'
    'ses'    $$000110000110    POSSESS_PK      '-lui'
    'leur'   $$110000110000    POSSESS_PK      '-les'
    'leurs'  $$000110000110    POSSESS_PK      '-les';
```

(recall that in dictionary words, the apostrophe is written ^). Thus, "son oiseau" means "his bird" or "her bird", according to the current meaning of "-lui", i.e., according to the gender of the most recent singular noun referred to.

The parser automatically tries both meanings if the same word occurs in both pronoun and descriptor tables. This happens in English, where "her" can mean either a feminine singular possessive adjective ("take her purse") or a feminine singular object pronoun ("wake her up").

(II.4) Informese vocabulary: numbers

An array should be given of words having type ⟨number⟩. These should include enough to express the numbers 1 to 20, as in the example:

```
Array LanguageNumbers table
    'un' 1 'une' 1 'deux' 2 'trois' 3 'quatre' 4 'cinq' 5
    'six' 6 'sept' 7 'huit' 8 'neuf' 9 'dix' 10
    'onze' 11 'douze' 12 'treize' 13 'quatorze' 14 'quinze' 15
    'seize' 16 'dix-sept' 17 'dix-huit' 18 'dix-neuf' 19 'vingt' 20;
```

In some languages, such as Russian, there are numbers larger than 1 which inflect with gender: please recognise all possibilities here. If the same word appears in both numbers and descriptors tables, its meaning as a descriptor takes priority, which is useful in French as it means that the genders of "un" and "une" are recognised after all.

(III.1) Translating natural language to Informese

Part III holds the routine to convert what the player has typed into Informese. In "English.h" this does nothing at all:

```
    [ LanguageToInformese; ];
```

This might just do for Dogg, the imaginary language in which Tom Stoppard's play *Dogg's Hamlet* is written, where the words are more or less English words rearranged. (It begins with someone tapping a microphone and saying "Breakfast, breakfast... sun, dock, trog...", and "Bicycles!" is an expletive.) Other languages are structurally unlike

English and the task of `LanguageToInformese` is to rearrange or rewrite commands to make them look more like English ones. Here are some typical procedures for `LanguageToInformese` to follow:

(1) Strip out optional accents. For instance, `"German.h"` looks through the command replacing ü with `ue` and so forth, and replacing ß with `ss`. This saves having to recognise both spelling forms.

(2) Break up words at hyphens and apostrophes, so that:

> donne-lui l'oiseau *give the bird to him* ↦ donne -lui l' oiseau

(3) Remove inflections which don't carry useful information. For instance, most German imperatives can take two forms, one with an 'e' on the end: "lege" means "leg" (German: "put") and "schaue" means "schau" (German: "look"). It would be helpful to remove this "e", if only to avoid stuffing game dictionaries full of essentially duplicate entries. (Toni Arnold's `"German.h"` goes further and strips all inflections from nouns, while Ralf Herrmann's preserves inflections for parsing later on.)

(4) Break affixes away from the words they're glued to. For instance:

> della *of the* (Italian) ↦ di la
> cogela *take it* (Spanish) ↦ coge la

so that the affix part "la" becomes a separate word and can be treated as a pronoun.

(5) Replace parts of speech not existing in Informese, such as pronominal adverbs, with a suitable alternative. For instance:

> dessus *on top of it* (French) ↦ sur lui
> dedans *inside it* ↦ dans lui

(6) Alter word order. For instance, break off an enclitic and move it between two nouns; or if the definite article is written as a suffix, cut it free and move it before the noun:

> arma virumque *arms and the man* (Latin) ↦ arma et virum
> kakane *the cakes* (Norwegian) ↦ ne kake

When the call to `LanguageToInformese` is made, the text that the player typed is held in a `->` array called `buffer`, and some useful information about it is held in another array called `parse`. The contents of these arrays are described in detail in §2.5.

The translation process has to be done by shifting characters about and altering them in `buffer`. Of course the moment anything in `buffer` is changed, the information in `parse` becomes out of date. You can update `parse` with the assembly-language statement

```
    @tokenise buffer parse;
```

(And the parser does just this when the `LanguageToInformese` routine has finished.) The most commonly made alterations come down to inserting characters, often spaces, in between words, and deleting other characters by overwriting them with spaces. Inserting characters means moving the rest of `buffer` along by one character, altering the length `buffer->1`, and making sure that no overflow occurs. The library therefore provides a utility routine

```
    LTI_Insert(position, character)
```

to insert the given character at the given position in `buffer`.

• **EXERCISE 110**
Write a `LanguageToInformese` routine to insert spaces before each hyphen and after each apostrophe, so that:

donne-lui l'oiseau	*give the bird to him*	$\mapsto$	donne -lui l' oiseau

• **EXERCISE 111**
Make further translations for French pronominal adverbs:

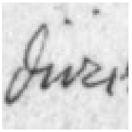
dessus	*on top of it*	$\mapsto$	sur lui
dedans	*inside it*	$\mapsto$	dans lui

• △**EXERCISE 112**
Write a routine called `LTI_Shift(from,chars)`, which shifts the tail of the `buffer` array by `chars` positions to the right (so that it moves leftwards if `chars` is negative), where the "tail" is the part of the buffer starting at `from` and continuing to the end.

• △**EXERCISE 113**
Write a `LanguageToInformese` routine which sorts out German pronominal adverbs, which are made by adding "da" or "dar" to any preposition. Beware of breaking a name like "Darren" which might have a meaning within the game, though, so that:

davon	$\mapsto$	von es
darauf	$\mapsto$	auf es
darren	$\not\mapsto$	ren es

§37 Names and messages in non-English languages


The fourth and final part of the language definition file is taken up with rules on printing out messages and object names in the new language. The *gender-number-animation (GNA)* combination is considerably more important when printing nouns than when parsing them, because the player is less forgiving of errors. Few errors are as conspicuous or as painful as "You can see a gloves here.", in which the library's list-writer has evidently used the wrong GNA for the gloves. Here is a more substantial example:

> *Volière*
> Une jungle superbe, avec des animaux et des arbres.
> On peut voir ici trois oiseaux (une oie, un moineau et un cygne blanc), cinq boîtes, une huître, Edith Piaf et des raisins.

To print this, the list-writer needs to know that "oie" is feminine singular, "cygne blanc" is masculine singular and so on. In short, it must be told the GNA of every object name it ever prints, or it will append all the wrong articles.

The translator will need first to decide how the genders are to be used. Inform allows for three genders, called `male`, `female` and `neuter` because they are usually used for masculine, feminine and neuter genders. Different natural languages will use these differently. In English, all nouns are neuter except for those of people (and sometimes higher animals), when they follow the gender of the person. Latin, German and Dutch use all three genders without any very logical pattern, while French, Spanish and Italian have no neuter. In Norwegian even the number of genders is a matter of dialect: traditional Norwegian has two genders, "common" and "neuter", but more recently Norwegian has absorbed a new feminine gender from its rural dialects. One way to achieve this in Inform would be to use `male` for common, `female` for the rural feminine and `neuter` for neuter. To avoid confusion it might be worth making the definition

```
Attribute common alias male;
```

which makes `common` equivalent to writing `male`. (The keyword `alias` is used, though very seldom, for making alternative names for attributes and properties.)

Here's how the library determines the GNA of an object's short name. The A part is easy: all objects having the `animate` attribute are animate and all others are inanimate. Similarly for the N part: objects having `pluralname` are plural, all others singular. (An object having `pluralname` is nevertheless only one object: for example an object called "doors" which represents a pair of doubled doors, or "grapes" representing a bunch of grapes.) If the object has `male`, `female` or `neuter` then the short name has masculine, feminine or neuter gender accordingly. If it has none of these, then it defaults to the gender `LanguageAnimateGender` if animate and `LanguageInanimateGender` otherwise. (These are constants set by the language definition file: see (IV.1) below.) You can find the GNA associated with an object's short name by calling the library routine

```
    GetGNAOfObject(obj);
```

which returns the GNA number, 0 to 11.

• **EXERCISE 114**
Devise a verb so that typing "gna frog" results in "frog: animate singular neuter (GNA 2) / The frog / the frog / a frog", thus testing all possible articled and unarticled forms of the short name.

. . . . .

In some languages, though not English, short names are inflected to make them agree with the kind of article applied to them:

das rote Buch	*the red book* (German)
ein rotes Buch	*a red book*

In printing as in parsing, the library variable `indef_mode` holds `true` if an indefinite article attaches to the noun phrase and `false` otherwise. So one rather clumsy solution would be:

```
    Object Buch
      with ...
           short_name [;
             if (indef_mode) print "rotes Buch"; else print "rote Buch";
             rtrue;
           ];
```

In fact, though, the library automatically looks for a `short_name_indef` property when printing short names in indefinite cases, and uses this instead of `short_name`. So:

```
    Object Buch
      with short_name "rote Buch", short_name_indef "rotes Buch";
```

An automatic system for regular inflections of short names is possible but not easy to get right.

In languages other than English, short names also inflect with case, and the best way to handle this may be to provide new printing rules like `dative_the`, enabling the designer to write code like so:

```
          "You give ", (the) noun, " to ", (dative_the) second, ".";
```

(IV.1) Default genders and contraction forms

Part IV of a language definition file opens with declarations of the default gender constants mentioned above. "English.h" has

```
Constant LanguageAnimateGender   = male;
Constant LanguageInanimateGender = neuter;
```

whereas French would define both to be male.

Inform uses the term *contraction form* to mean a textual feature of a noun which causes any article in front of it to inflect. English has two contraction forms, "starting with a vowel" and "starting with a consonant", affecting the indefinite article:

a + orange = an orange
a + banana = a banana

The first constant to define is the number of contraction forms in the language. In the case of "French.h" there will be two:

```
Constant LanguageContractionForms = 2;
```

Of these, form 0 means "starting with a consonant" and 1 means "starting with a vowel or mute h". (It's up to you how you number these.) You also have to provide the routine that decides which contraction form a piece of text has. Here is an abbreviated version for French, abbreviated in that it omits to check accented vowels like 'é':

```
[ LanguageContraction text;
  if (text->0 == 'a' or 'e' or 'i' or 'o' or 'u' or 'h' or
                 'A' or 'E' or 'I' or 'O' or 'U' or 'H') return 1;
  return 0;
];
```

The text array holds the full text of the noun, though this routine would normally only look at the first few letters at most. The routine is only ever called when it is necessary to do so: for instance, when the library prints "the eagles", LanguageContraction is not called because the article would be the same regardless of whether "eagles" has contraction form 0 or 1.

• **EXERCISE 115**

Italian has three contraction forms: starting with a vowel, starting with a 'z' or else 's'-followed-by-a-consonant, and starting with a consonant. Write a suitable LanguageContraction routine.

(IV.2) How to print: articles

English needs two sets of articles: one set for singular nouns, which we shall call article set 0, another for plurals, article set 1. We need to define an array to show which GNAs result in which article set:

```
                       !             a           i
                       !             s     p     s     p
                       !             m f n m f n m f n m f n
    Array LanguageGNAsToArticles --> 0 0 0 1 1 1 0 0 0 1 1 1;
```

(The number of article sets is not defined as a constant, but instead by the contents of this array: here the only values are 0 and 1, so there need to be two article sets.) We also need to define the article sets themselves. There are three articles for each combination of contraction form and article set. For example, "English.h" has two contraction forms and two article sets, so we supply twelve articles:

```
    Array LanguageArticles -->
      !   Contraction form 0:     Contraction form 1:
      !   Cdef   Def    Indef     Cdef   Def    Indef
          "The " "the " "a "      "The " "the " "an "      ! Set 0
          "The " "the " "some "   "The " "the " "some ";   ! Set 1
```

That defines the automatic rules used to apply articles to nouns, but there are two ways to override this: the property `article`, if present, specifies an explicit indefinite article for an object; and the property `articles`, if present, specifies an explicit set of three articles. This is useful for nouns whose articles are irregular, such as the French "haricot": the regular definite article would be "l'haricot", but by an accident of history "le haricot" is correct, so:

```
    Object "haricot"
      with articles "Le " "le " "un ", ...
```

• **EXERCISE 116**
Construct suitable arrays for the regular French articles.

• **EXERCISE 117**
Likewise for Italian, where Inform needs to be able to print a wider selection: un, un', una, uno, i, il, gli, l', la, le, lo.

• **EXERCISE 118**
At the other extreme, what if (like Latin: "vir" *man* or *a man* or *the man*) a language has no articles?

(IV.3) How to print: direction names

Next is a routine called `LanguageDirection` to print names for direction properties (*not* direction objects). Imitate the following, from `"French.h"`:

```
[ LanguageDirection d;
  switch (d) {
      n_to:    print "nord";      s_to:    print "sud";
      e_to:    print "est";       w_to:    print "ouest";
      ne_to:   print "nordest";   nw_to:   print "nordouest";
      se_to:   print "sudest";    sw_to:   print "sudouest";
      u_to:    print "haut";      d_to:    print "bas";
      in_to:   print "dans";      out_to:  print "dehors";
      default: RunTimeError(9,d);
  }
];
```

(IV.4) How to print: numbers

Next is a routine called `LanguageNumber` which takes a number `N` and prints it out in textual form. `N` can be anything from `-32767` to `32767` and the correct text should be printed in every case. In most languages a recursive approach makes this routine less enormous than it might sound.

• **EXERCISE 119**
Write `LanguageNumber` for French.

(IV.5) How to print: the time of day

Even mostly numeric representations of the time of day vary from language to language: when it's 1:23 pm in England, it's 13h23 in France. A routine called `LanguageTime-OfDay` should print out the language's preferred form of the time of day, like so:

```
[ LanguageTimeOfDay hours mins;
  print hours/10, hours%10, "h", mins/10, mins%10;
];
```

• **EXERCISE 120**
Write the corresponding English version.

(IV.6) How to print: verbs

The parser sometimes needs to print verbs out, in messages like:

> I only understood you as far as wanting to *take* the red box.

It normally does this by simply printing out the verb's dictionary entry. However, dictionary entries tend to be cut short (to the first 9 letters or so) or else to be abbreviations (rather as “i” means “inventory”). In your language, verbs might also need to inflect in a sentence like the one above, which assumes that the infinitive and imperative are the same. You might get around that by rewording the statement as:

> I only understood you as far as “*take* the red box”.

Even so, how to print out verbs depends on the language, so you need to give a routine called `LanguageVerb` which looks at its argument and either prints a textual form and returns true, or returns false to let the library carry on as normal. In English, only a few of the more commonly-used abbreviations are glossed, and “x” for “examine” is the only one that really matters:

```
[ LanguageVerb verb_word;
  switch (verb_word) {
      'l//': print "look";
      'z//': print "wait";
      'x//': print "examine";
      'i//', 'inv', 'inventory': print "inventory";
      default: rfalse;
  }
  rtrue;
];
```

(IV.7) How to print: menus

Next, a batch of definitions should be made to specify the look of menus and which keys on the keyboard navigate through them. Imitate the following `"English.h"` definitions, if possible keeping the strings the same length (padding out with spaces if your translations are shorter than the English original):

```
Constant NKEY__TX     = "N = next subject";
Constant PKEY__TX     = "P = previous";
Constant QKEY1__TX    = "  Q = resume game";
Constant QKEY2__TX    = "Q = previous menu";
Constant RKEY__TX     = "RETURN = read subject";
Constant NKEY1__KY    = 'N';
Constant NKEY2__KY    = 'n';
Constant PKEY1__KY    = 'P';
Constant PKEY2__KY    = 'p';
Constant QKEY1__KY    = 'Q';
Constant QKEY2__KY    = 'q';
```

(IV.8) How to print: miscellaneous short messages

These are phrases or words so short that the author decided they probably weren't worth putting in the `LibraryMessages` system (he now thinks otherwise: code in haste, repent at leisure). Here are some French versions with notes.

```
Constant SCORE__TX     = "Score: ";
Constant MOVES__TX     = "Tours: ";
Constant TIME__TX      = "Heure: ";
```

which define the text printed on the status line: in English, "Score" and "Turns" or "Time";

```
Constant CANTGO__TX    = "On ne peut pas aller en ce direction.";
```

the default "You can't go that way" message;

```
Constant FORMER__TX    = "votre m@^eme ancien";
```

the short name of the player's former self, after the player has become somebody else by use of the `ChangePlayer` routine;

```
Constant YOURSELF__TX = "votre m@^eme";
```

the short name of the player object;

```
Constant DARKNESS__TX = "Obscurit@'e";
```

the short name of a location in darkness;

```
Constant NOTHING__TX   = "rien";
```

the short name of the `nothing` object, caused by `print (name) 0;`, which is not strictly speaking legal anyway;

```
Constant THAT__TX      = "@cca";
Constant THOSET__TX    = "ces choses";
```

(`THOSET` stands for "those things") used in command printing. There are three circumstances in which all or part of a command can be printed by the parser: for an incomplete command, a vague command or an overlong one. Thus

```
>take out
What do you want to take out?
>give frog
(to Professor Moriarty)
```

```
>take frog within cage
```
I only understood you as far as wanting to take the frog.

In such messages, the `THOSET__TX` text is printed in place of a multiple object like "all" while `THAT__TX` is printed in place of a number or of something not well understood by the parser, like the result of a `topic` token.

```
Constant OR__TX        = " ou ";
```

appears in the list of objects being printed in a question asking you which thing you mean: if you can't find anything grammatical to go here, try using just `", "`;

```
Constant AND__TX       = " et ";
```

used to divide up many kinds of list;

```
Constant WHOM__TX      = "qui ";
Constant WHICH__TX     = "lequel ";
Constant IS2__TX       = "est ";
Constant ARE2__TX      = "sont ";
```

used only to print text like "inside which is a duck", "on top of whom are two drakes";

```
Constant IS__TX        = " est";
Constant ARE__TX       = " sont";
```

used only by the list-maker and only when the `ISARE_BIT` is set; the library only does this from within `LibraryMessages`, so you can avoid the need altogether.

(IV.9) How to print: the Library Messages

Finally, Part IV contains an extensive block of translated library messages, making up at least half the bulk of the language definition file. Here is the entry for a typical verb in `"English.h"`:

```
SwitchOn:
    switch (n) {
        1: print_ret (ctheyreorthats) x1,
               " not something you can switch.";
        2: print_ret (ctheyreorthats) x1,
               " already on.";
        3: "You switch ", (the) x1, " on.";
    }
```

You have to translate every one of these messages to at least a near equivalent. It may be useful to define new printing rules, just as "English.h" does:

```
[ CTheyreorThats obj;
  if (obj == player) { print "You're"; return; }
  if (obj has pluralname) { print "They're"; return; }
  if (obj has animate)
  {   if (obj has female) { print "She's"; return; }
      else if (obj hasnt neuter) { print "He's"; return; }
  }
  print "That's";
];
```

• **EXERCISE 121**
Write a printing rule called `FrenchNominativePronoun` which prints the right one out of il, elle, ils, elles.

• **REFERENCES**
Andreas Hoppler's alternative list-writing library extension "Lister.h" is partly designed to make it easier for inflected languages to print out lists.