Text Based Interactive Fiction and Computational Linguistics

Toni Arnold
Seestrasse 369
8038 Zürich
tarnold@cl.unizh.ch

Abstract

Interactive fiction (IF) or text adventures are text-based computer games. After a short introduction I will explain some details of an IF authoring system. As an illustration I will give an outline of my own project. I will conclude by discussing some scientific attempts to improve the genre using theories of artificial intelligence and computational linguistics.

1 What is interactive fiction (IF) ?

"IF means a lot of different things to different people. (...) IF is an artform, a work of love. I use the term to refer to text adventures exclusively. So, IF is to graphic games what books are to Network television. They are geared towards a more cerebral audience. The people that write IF usually have a deep-set love for language and its nuances. We are not satisfied with a few little mouse icons as a user interface. Instead we try to harness as much natural language as we can." [Wilson 94: 2]

A text adventure consists of a parser that understands the input typed by the player and a considerable amount of text output which tells the player about the story, gives descriptions of places and objects and the effects of his/her actions. The input usually has the form of imperative sentences which cause an actor in the story to change the state of a model world. In many games it is possible to communicate with other actors. The player usually achieves the goal of the game by solving puzzles; this often takes much time. For that reason the current state of the game can be saved to disc and later restored.

Like most human inventions IF was developed by simulating nature. A physicist, Patricia Crowther, found a hidden way into the Mammoth Cave, a huge cave in the Kentucky karst. "Patricia's husband, Willie Crowther, wrote a computer simulation of his favourite region, Bedquilt Cave, in FORTRAN in the early 1970s. (...) A colleague of the Crowthers (at a Massachusetts computing firm), Don Woods, stocked up the caves with magical items and puzzles, inspired by a role-playing game." [Nelson 95: 3] The result, the first text adventure named "Colossal Cave", can still be played. It is available at the if-archive (for the URL see the references). In the 80s, the two biggest companies, Magnetic Scrolls (UK) and Infocom (USA) had some commercial success, but in the 90s the genre disappeared from the software stores - it was not profitable enough. The explanation of Wilson is as follows: "My personal theory is that each new generation is being raised with more and more emphasis placed on visual stimulus in the form of television, WINDOWS 3.1, etc. This carries over into their recreation, and so, we are left with a smaller percentage of the populace each year that is literate enough to enjoy a good text adventure and understand the references made in them. C'est la vie." [Wilson 94: 2] He dedicates his guide to the memory of Infocom. "Although they live on, it will never be the same."

2 The Authoring Systems

Thanks to the internet, IF seems to become popular again. There are many authoring systems available at the if-archive. The "Which Authoring System is Better" FAQ [Newell 96] shows a useful comparison of ten different systems. Most of them are available for most of the contemporary platforms (except of course your own, as usual), but sometimes the various implementations differ in stability. Such a system usually consists of a compiler designed for a special IF programming language and a runtime interpreter. The compiler generates platform-independent code which is executed by the interpreter. With this technique it is easy to achieve a maximum of portability; but there are also some compilers which create stand-alone applications for the system they run on.

On the one hand there are systems like ALAN which try to faciliate the writing of a text adventure as much as possible. "ALAN has a syntax which is very much declarative English (or Swedish) in format. There is an object/block structure which is easy to understand, and coding within each block is mostly uncomplicated." [Newell 96: 23] The main problem with these systems is that it is difficult or even impossible to extend them. On the other hand there are systems like Inform or TADS which are not as easy to learn because they use a syntax similar to C++ - but they offer more possibilities for improvement. The most elaborated ones are the Inform-compiler with its libraries by Graham Nelson and the TADS (Text Adventure Developing System). For the TADS David Bagget has written an alternative library named WorldClass which has been documented by Paul Gilbert. This documentation is a good introduction to an IF-specific model world. [Gilbert 94]

TADS mixes up library programming code and text that is printed on the screen. This makes it very difficult to translate into another language. Inform separates these two parts as much as possible. A translator only has to change the grammar and the library messages object, not the library itself. Recently, Tinic Urou has provided a German library for Inform based on the English original.

3 Inform - Some Details

In this section I will explain briefly how an IF programming system works. I chose Inform as example because it is the system I am working with. For more details see Graham Nelson's Inform designers manual [Nelson 96] or Gareth Rees' Inform Tutorial [Rees 95]. The user interface is a command line interpreter for natural language. The input is cleared up and sent to a parser. The Inform parser first recognizes the type of the nominal phrases to be able to disambiguate the verb by some kind of selectional restrictions. It tries to match the result with grammar rules like the following:

```
Verb "drop" "throw" "discard"
        * multiheld                     -> Drop
        * multiexcept "into" noun       -> Insert
```

```
* held "at" noun                -> ThrowAt;
```

The verbs after `Verb` are near-synonyms. The imperative verb is the first word in the user input. The quoted words, usually prepositions, must occur literally at the indicated place in the sentence, the others are nominal phrases. The parser now tries to set up the variables 'action' for the logical action, 'object' for the direct object and 'second' for the indirect object. Objects can only be referred to if they are 'InScope' which means either that the player holds them or that they are visible for him/her in the current room. `multiheld` means that there is one or more objects that need to be held by the actor. Then the logical action 'Drop' is executed. The second line matches sentences like "Throw every fish except the dead fish into the river". `multiexcept` is a construction which the parser recognizes as one nominal phrase denoting a set of objects.

At this stage, the Inform parser can put questions to the user like "Which do you mean, the red river or the black river?" if the reference is ambiguous because the player stands at the intersection of two rivers. The library does not distinguish between nouns and adjectives. They are all taken as names denoting an object. Nelson gives an example in his manual how adjectives could be implemented properly. If the player responds "black river" or just "black", the black river is considered to be 'second'. If this construction matches the game's requirements, "throw" means logically 'Insert' which makes the 'held' objects children of the object in 'second' in the internal object tree.

If the player types "Throw the stone at the policeman" the action is recognized as 'ThrowAt' if the player holds the object denoted by "stone". This might cause damage to the 'noun' object or invoke some reaction. After the action, the 'held' object - the stone - lies on the floor. This results in changes in the object tree: the stone is moved from its position as a *child* of the player object to be a *child* of the current room object. It thus becomes a *sister* to the player object and the policeman object.

Inform does anaphora resolution by setting up variables for "it", "him" and "her" after every input. 'itobj' always holds the last object, 'himobj' the last male person and 'herobj' the last female person mentioned. If there is no object in the current user input, the anaphora variables remain unchanged.

The actions are defined in the library. Logically, most of an IF game is defined in terms of exceptions to the library default actions. Inform provides a before/after system to change these defaults. The policeman object can own a 'before' routine which is performed before the library action is executed. One might insert a little routine that prompts the player to answer if he/she really wants to do certain things. The 'after' routine could start a 'daemon', a routine that is called after each turn of the player. Causality is often implemented using the before/after system: e.g. the 'after' routine of the policeman object causes the policeman object to shoot. Sometimes causal relations are hidden in the procedural code - for example killing animate objects by shooting them is hidden in the 'daemon' routine of the policeman object.

Inform tries to execute the 'before' routines owned by objects in a fixed order: first the player, then the current location, then the 'noun' object. If one of these routines returns 'true', Inform stops and prompts the player for the next turn. In the policeman example this would happen when the player answers "no" to the warning. If all 'before' routines return 'false', the system tries to perform the desired action in the model world. After a successful action, the 'after' routines are called in the same order. If one of them returns true, perhaps after printing "You hit the policeman with the stone. He seems to be real angry now.", Inform stops and the default message is omitted.

The model world of Inform consists of objects which can be defined by a class hierarchy with inheritance of properties. If the system tries to carry out an action, it infers by means of these properties if it is possible or not. Of course the game objects can override these defaults with their own definitions.

The effects of an action are finally printed on the screen in natural language. The output is not submitted to a grammar. Inform can only generate nominal phrases from objects. The programmer writes lines such as

```
print "You throw ", (a) noun, " into ", (the) second, ". It disappears."
```

When printing, '(the) second' is replaced by the indirect object of the input sentence with the definite article and its name - in the example above with "the black river". '(a) noun' is replaced by the first living fish held by the player. There is no agreement between noun and indefinite article ("a fish" vs "an elephant"). If there is more than one 'noun' object, the sentence is repeated for every object thrown into the river. After the output has finished, the player is prompted for the next turn.

4 An IF System in Prolog

My attempt to write an IF authoring system resulted in a 100K Prolog toy program with a small library. The idea was to implement a system that allows writing a game in natural language just like a story. This story is interpreted by the compiler and transformed into an executable game.

I have implemented a small LFG-like definite clause grammar (DCG) which parses the user input. The main advantage of LFG (Lexical Functional Grammar) compared to other syntactic theories is that it is easy and fast to implement. The speed factor is extremely important because a player does not want to wait minutes for the systems's answer. For a full-size system the only way to achieve such short response times is probably to do partial parsing. This means that the parser only looks for nominal phrases, prepositions and verbs (for example), but does not try to parse the whole sentence.

The derived functional structure is a feature structure containing the words of the user input with their grammatical functions. I matched the functional structure of the input with partially instantiated functional structures belonging to the logical actions. The functional structure is only one part of the output of an LFG-parser, the other part is a conventional syntax tree called constituent structure which I did not use for the semantic part. This construction allows the handling of free constituent order in languages like German because surface constituent order takes only effect on the constituent structure tree, not on the functional structure. If the system is designed in this manner, the selectional restrictions for ambiguity resolution and the resolution of nominal phrases have to be specified separately from the grammar. I still have to work on this problem.

I first tried to do output via the DCG-LFG-grammar, but the problems could not be solved at this stage of the project. Afterwards I was learning that LFG was never meant to be able to generate sentences. In a good system, the input parser should understand grammatically incomplete sentences like "take key, put it in pocket", but it must create only correct sentences for the output. For this reason it is not suitable to use the same grammar for input and output. A linguistically satisfactory system should do morphology, too. For the English language it is easy to implement an NLP system without morphology analysis; for German it is possible; for heavily inflected languages like Estonian it is senseless. Again, the input morphology should differ from the output morphology. Output words must be correct, input words should be recognized even when they are misspelled.

Every text adventure system must be able to save the state of the model world. The underlying machine of the Inform system, the 'Z-Machine', was designed to be able to save its state of execution and its variables at any time. In Prolog it is hardly possible to save variable bindings, but it is easy to write or read clauses to/from disc. In my experiment, I used two types of clauses, some marked with static_(Clause) and some marked with dynamic_(Clause). Static clauses define, for example, the rooms or unmovable objects and their description texts. Dynamic clauses refer to items like the current location of the player and other movable objects, the state of some objects or the actual score. If the player restores a saved game, all dynamic clauses are retracted. Then the saved file with a listing of all dynamic clauses at the time of saving is consulted into memory. If the player starts a new game, an initial state is asserted. If the player takes up another game, all static and dynamic clauses are retracted and the new game is consulted from disc. An Inform-like before/after system is easy to implement in Prolog. The core are clauses called with the parser output as input of the form

```
do(F_structure):-
   extract(F_structure, Action, Object),
   try_before(Action, Object),
   act(Action, Object),
   try_after(Action, Object), !.
```

and the following clauses:

```
try_before(Action, Object) :- \+ clause( before(Action, Object), _).
try_before(Action, Object) :- before(Action, Object).
```

Somewhere in the game code there could be a 'before' clause like the following ('try_after' clauses are defined in the same way as 'try_before' clauses and 'after' clauses work the same way as 'before' clauses):

```
before(throw, stone) :-
  write('Are you really sure about what you are doing? '),
  read(yes).
```

The example is executed by calling `?- do(X).` if you add:

```
extract(_, throw, stone).
act(throw, stone) :- write('Stone thrown!'), nl.
```

The 'do' clause first tries to prove 'try_before'. If it fails to find a 'before' clause for the desired action and object, it returns 'true' because of the negation '`\+`', and Prolog continues proving 'act(...)'. If there is such a 'before' clause, the first 'try_before' clause fails and the second calls the particular 'before' clause of the action and the object.

In my little example the 'before' clause succeeds when the player answers 'yes'. Then 'try_before' succeeds and the action is carried out. If the player answers 'no', 'before' and consequently 'try_before' fails. Prolog backtracks and the action is not carried out. With the exception that 'before' should return 'fail' instead of 'true' if the action is to be omitted, the principle is the same as in the Inform system - but in pure Prolog, without any extensions.

5 Specifying an IF Game in Natural Language

The primary idea for my system was to specify a game in natural language. The system compiles the input text which means that it translates it into Prolog clauses marked as static_(Clause). I have not been able to do this for verbs, but with the help of some hacks my little toy worked for the objects. The programmer must provide a text file like the following (my system is written in German, so the following is a translation of the original):

> The player starts in the street. The street writes: [You are in the street. It is raining.] The player has the key. The garden is north of the street. The dog is in the garden. The dog writes: [This is the dog that your girl friend has left you after leaving you. It is barking, as usual.] The entry is east of the garden. The door is between the entry and the garden. The door is closed. The door is locked with the lock. The key opens the lock.

The texts in brackets are not interpreted by the parser. They are just printed out by the system as descriptions of objects for the player to read.

A big problem is the resolution of **presuppositions**. When the programmer writes the sentence "The garden is north of the street." he/she probably (!) means: "There is a room object called garden. There is a room object called street. The garden is north of the street. The street is south of the garden." If there is no handling of presuppositions no author will ever use the system because the source text becomes much too large and awkward.

I did not implement any debugging features yet. The compiler only reports what it is doing. Playing with my system I found out that the compilation of a natural language source has to be done interactively with the author. For example when the author spells an existing room wrongly, the system takes that as presupposition for a new room and creates it. The programmer has to recognize the mistake, to change the source text and to recompile. This is annoying.

If there are ambiguities or strange presuppositions the compiler should not only tell the user what it understands. Instead it should prompt him/her to confirm or change for example some presuppositions. The system should replace sentences by unambigous sentences or add disambiguation remarks at the right place in the source code. If it's compiled again, the system will not prompt the user again. This is only possible by aligning the source text with the derived Prolog clauses. If this is done properly, Prolog allows to change the program at runtime. The IF author can modify little things and immediately see the effects on the game - a feature that no existing authoring system provides.

With this in mind I decided to give up the concept of a separate game compiler. Instead the 'compiler' will be a privileged actor in the game, for example Merlin working in his alchemist laboratoy. There could be some helper actors, perhaps a rat sitting on his shoulder commenting everything Merlin says - presupposition, anaphora resolution and so on. A parrot could imitate everything Merlin has said and create a golden egg containing the source code of the game Merlin is working on. An intelligent dog could do file handling by carrying the eggs around in his mouth and archiving them. And so on.

The core idea of that scenery is that in a Prolog system there is logically no difference between the compiler and the actors: both assert and retract Prolog clauses. A conventional actor in the game only has access to clauses marked as dynamic_(Clause). If the player becomes Merlin, he and his helpers get access to the whole Prolog code. Merlin can do everything a player actor can do, but it treats declarative sentences different. If Merlin says "There is light in the bedroom." then there is light in the bedroom - the new fact is accommodated to the model world. If a player actor said the same sentence, the system would interpret that as a question and answer whether it is true or not.

The author doesn't have to leave the game to write it. Only for changes, which Merlin with his helpers can't do because nobody has implemented it yet, the author must dive into Prolog coding. If this was possible, a game could be bootstrapped in natural language - this means that a game could be written with itself. Such an enhanced game system should become very general. Perhaps it could even be reused for writing question answering systems about a natural language database by giving it to Merlin to read.

5.1 IF and AI

After coding my little toy, I learned that Norbert E. Fuchs and Rolf Schwitter of the Department of Computer Science at the University of Zurich are working on a project to specify a Prolog program using natural language: "The present prototypical implementation of Attempto proves that Attempto Controlled English (ACE) can be used for the non-trivial specification of an automated teller machine, and that the complete specification can be translated as coherent text into a structured form of first-order logic - and optionally into Prolog clauses - that can be executed." [Fuchs & Schwitter 96] They use a DCG grammar enhanced by feature structures. For the resolution of anaphora and other inter-textual references they use Discourse Representation Theory.

This proves that Prolog is a good choice for programming an enhanced interactive fiction authoring system. The main problem using this AI-language is its portability which depends on the availability of good public domain Prolog interpreters or even compilers. At the moment, the situation can be problematic. The author has to divide his/her code into a portable part which is written in standard Edinburgh syntax and a little interface part with a code for any system on which he/she wants to run his/her game. The player has to select the system before the consulting process. Especially file handling, terminal input/output and some other built-ins differ from system to system. I wrote my toy system using Open Prolog for Macintosh written by Michael Brady, which is public domain (for the URL see the references).

The main problem using AI programming knowledge is the speed of the implementations. For scientific purposes, the running speed of a program is often irrelevant. But as I mentioned before, an IF system should respond in less than a second. Only for actions that the player him/herself recognizes as very complicated, it may be allowed to take a little bit more time - but only a little bit. There are some scientific projects to create authoring systems for IF. The people of the Oz project are developing a system that allows authors to create and present interactive dramas, but I have never seen a game designed with this system. They try to simulate the psyche of the actors, but the cat example that they claim to be a believable actor doesn't convince me at all. An overview and some publications are available over the internet (for the URL see the references).

I believe that efforts to provide complete world knowledge stem from a deep-set misconception of the whole IF project: If we are talking about art, it is not the computer that has to provide good art, and it is not the engineer either. It is the artist's imagination which makes a little piece of world appear in a light never seen before. This is what people want to experience when they dedicate their time to the artist's product. I think that it will never be possible to provide one single system that every IF author would like to use - artists are known to have their own minds. I also think that writing a text adventure will always be closer to writing a novel or a theatre play than to doing AI-engineering.

It is certainly possible to implement formalized sociological or psychological theories on a computer and to use them for interactive fiction; and certainly the attempt of using them to transfer human knowledge onto machines is very interesting. However, I think that if there is a computerized implementation of human beings or whole societies, we demonstrate that humans behave like a machine; that they are just an instantiation of a role which can be written down explicitly. I therefore believe that if the genre wants to be taken serious as an artform, it has to make the audience experience the laughter of thousands of years of text-based human history about these silly modern toys called computers - not despite but because of the fact that we love those magic machines!

6 References

The main archive for interactive fiction with games, authoring systems and documentations is located at ftp://ftp.gmd.de/if-archive. The URL's beginning with ".../" refer to the if-archive.

6.1 Literature

- Fuchs, Norbert E. & Schwitter, Rolf: Attempto Controlled English (ACE), CLAW 96, The First International Workshop on Controlled Language Applications, 26-27 March 1996.
  ftp://ftp.ifi.unizh.ch/pub/techreports/other_docs/CLAW96.ps.gz
- Gilbert, Paul: The WorldClass System Author's manual, 1994.
  .../programming/tads/worldclass/paulslib.zip
- Nelson, Graham: The craft of adventure, 1995.
  .../info/Craft.Of.Adventure.*
- Nelson, Graham: The Inform Designer's Manual, Oct. 1995 (95b).
  .../programming/inform6/manuals/designers_manual.*
- Newell, Bob: The "Which Authoring System is Better" FAQ, Feb. 1996.
  .../info/whichsys.zip
- Oz Project site: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/oz/web/index.html

- Rees, Gareth: An Inform tutorial, 1995.
- Wilson, G. Kevin: Whizzard's Guide to Text Adventure Authorship, v2.0, 1994. .../info/authorship-guide.base

6.2 Executables

- TADS: .../programming/tads/
- Inform: .../programming/inform6/
- German Inform:.../programming/inform6/library/german_inform_05.tar.gz
- Z-code games (platform independent): .../games/infocom/
- Z-code interpreters: .../infocom/interpreters/zip/
- various games: .../games/
- Open Prolog for Macintosh: ftp://ftp.cs.tcd.ie/pub/languages/open-prolog/OpenProlog.sea.hqx

6.3 Newsgroups

- rec.arts.int-fiction (primarily for discussing authorship)
- rec.games.int-fiction (more for IF-gamers; hints and such)
- rec.games.programmer (for anything considering game programming)