Available online at www.sciencedirect.com
SCIENCE DIRECT
Knowledge-Based Systems 18 (2005) 19–35
Knowledge-Based SYSTEMS
www.elsevier.com/locate/knosys

A natural language help system shell through functional programming

Robert Plant[a,*], Stephen Murrell[b]

[a]Department of Computer Information Systems, University of Miami, Coral Gables, FL 33124, USA
[b]Department of Electrical and Computer Engineering, University of Miami, Coral Gables, FL 33124, USA

Received 30 November 2003; accepted 21 April 2004
Available online 7 June 2004

Abstract

This paper investigates the development of a natural language (NL) interface for mixed initiative dialogues within a constrained domain and demonstrates the applicability of the functional approach to NL system development. The system consists of two major components, a natural language subsystem comprises a general-purpose parser that interprets a 'plug and play' tagged BNF grammar (which may be ambiguous), to parse natural language input and extract semantic information. The knowledge-based subsystem uses the semantic tags extracted by the natural language subsystem to generate a focused query to select the most appropriate script for a guided dialogue with the user. The system was written entirely in a purely functional language, which resulted in a surprisingly small and simple program.

Keywords: Natural language processing; Functional programming; Automated responses; Scripts

1. Introduction

With the deregulation of the Internet and the development of ever more complex and sophisticated systems such as enterprise resource planning, customer relationship management systems, OLAP systems and operating systems it was inevitable that a parallel requirement for more sophisticated online user assistance systems would follow. To this end, academic investigators and commercial developers have drawn upon diverse research areas including natural language understanding (NLU), ergonomics, database systems, human factors and knowledge-based systems to make their systems more intuitive, user friendly and intelligent. Resulting in the emergence of new technologies and platforms in several areas:

- Email handling systems, e.g. Jeeves Answers (www.jeevesolutions.com), e-Dialogue's Quick Reply (http://www.edialogue.com/) and ROI Direct.com's Customer Response (http://www.tele-direct.com/), being deployed by organizations such as the US Congress and Office Depot that receive large volumes of email correspondence.
- Natural-language web site search technology, e.g. Phrase technology's (www.iphrase.com), One-Step, utilized by Charles Schwab to enable its 75 million end users type simple natural language queries of its site.
- Call Center enhancement, e.g. InQuira (www.inquira.com) has developed a natural language (NL) interaction technology that helps simultaneously improve both the quality of the call center representatives' responses to customer queries but also to decrease the time taken to reach the solution, avoiding the decision tree type of analysis that is usually performed.
- Computer-based training systems that utilize natural language interaction have been receiving increased attention [1] and products such as Wex Tech's (http://www.wextech.com/kipr.htm) AnswerWorks 'question answering engine' are aimed at using natural language understanding to enhance web-based training.
- The use of NL queries of databases remains a difficult problem [2], however, researchers such as Owei, who developed a 'conceptual query language-with-natural language (CQL/NL)' [3] to assist in the filtering of natural language queries, have continued to develop solutions for subproblem categories.

All of these systems have at their core some form of computational natural language processing (NLP) system, an area that has a long history of investigation and research.

* Corresponding author. Tel.: +1-305-284-6105; fax: +1-305-284-5195.
E-mail address: rplant@miami.edu (R. Plant).

0950-7051/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.knosys.2004.04.002

The most basic technique for analyzing natural language is that of keyword matching, of which Weizenbaum's ELIZA [4,5] system is a well-known example. The major criticism of this type of NLP is that the dialogues generated tend to be very shallow and superficial, not allowing users to probe for solutions, involving, for example, inductive meta-knowledge. The second level of NLP techniques was inspired by Chomsky's work on grammars [6] whose work has radically influenced NLP. The grammars being used to 'parse' or break down the structure of the sentence helping establish their meaning, as opposed to keyword matchers, which are based upon the expectation of keywords being present in the sentence presented to them and with very little meaning being extracted from the input. There have been many types of grammars used within NL research including 'phrase structured grammars' [7] transformational grammars [6], case grammars [8] and syntactic grammars [9] and as such parsing is a central construct upon which NLP is based. Through the use of these grammatical rules in conjunction with other knowledge sources, the function of words within an input stream can be determined and the relations between them used to extract some degree of meaning from the sentences. Building upon the work of these early systems, researchers have taken a variety of approaches to building computational NLP systems including Refs. [10–22]. For a significant and comprehensive bibliography of texts in computational NLU refer to Mark Kantrowitz's 'Bibliography of Research in Natural Language Generation' [23], the survey from Varile and Zampolli [24] or the digital archive of research papers in computational linguistics at the University of Pennsylvania (http://www.cis.upenn.edu/~adwait/penntools.html).

This paper builds upon the previous research to present a new two-part computational NLP system based upon the use of an executable set of functional equations and through this notation demonstrates its applicability to the creation of knowledge-based online help systems. A prototype solution to the UNIX help assistant problem [13] is presented, this was felt to be a suitable domain through which the operational aspects of the approach could be tested as its solution space is formally defined, yet facilitates mixed initiative dialogues.

The natural language processing subsystem accepts as input a BNF description of the language. This approach has the advantage that the language module can be replaced or upgraded by any user who understands formal grammars without requiring any programming; it was largely abandoned in mainstream NLP research (partly) because of the ambiguous nature of natural languages. The approach taken here is to produce all possible parses of the input query. In the relatively restricted domain of help systems, input are not large: queries tend not to be multi-paragraph compositions, so although ambiguities may still produce more than one possible parse, the sometimes exponential explosion of possibilities is not a debilitating problem.

The BNF [25] accepted by the parser is extended with simple semantic tags that essentially say 'if a successful parse comes through here, make a note of xxxx'. The output from the parser is not just a list of possible parse trees, but also a list of possible tag sets. Each tag set contains the semantic tags that were encountered in an ultimately successful parse of the input. For example, in the famous example 'fruit flies like a banana', the word 'Flies' can be either a verb or a noun. In the extended BNF, where 'flies' is listed as a possible verb, the tag 'action-travel' could be specified; where it is listed as a possible noun, the tag 'actor-insect' could appear. With a similar treatment for 'fruit', 'like', and 'banana', the parser would produce two possible tag sets:

(actor-insect, actor-enjoy, object-food)
(actor-food, action-travel, manner-food)

Of course, the difference between subjects and objects can be noted by the same means. Associating semantic tags with a position in a parse tree rather than a linear lexeme stream allows aubtle but important distinctions of meaning to be discerned: 'the broken printer is grey' may deliver useful information, but is obviously not reporting a problem, but 'the grey printer is broken' requires a definite reaction, even though both have the same words used as the same parts of speech (pos). The disadvantage to this is that input can only be processed if it conforms exactly to the given grammar. The ability to handle ambiguity without any problems means that much more forgiving grammars can be used, and on-line help systems are more likely to be used by grammatically competent users, but the need for 'correct' input can not be completely ignored.

This simplified, linear representation of meaning would clearly not be sufficient for a full natural language understanding system, but in the restricted domain of on-line help systems with its much smaller expected inputs, it provides an appropriate level of detail for further analysis.

Given the list of possible tag sets it is often found that all of them are the same: ambiguities in parsing do not always reflect semantic ambiguities. When all the syntax has been discarded and the input reduced to a set of tags that represent it's meaning there will often be no ambiguity left. The matching agent searches through a knowledge base of scripts and selects those whose indexes most closely match the tag sets. The knowledge base is stored as an association list connecting scripts to sets of semantic tags that must be matched as closely as possibly. This is another process that helps to resolve ambiguity: similar but not identical tag sets may select the same script. If more than one script is still selected the user may be asked to clarify their meaning by selecting from the topic summaries associated with each script a simple 'Did you mean A or B—type question'), upon which a dialogue is entered into. The process is shown in Fig. 1.

The script selection process is a variation of the approach advocated by Hobbs (1995) for the creation of generic

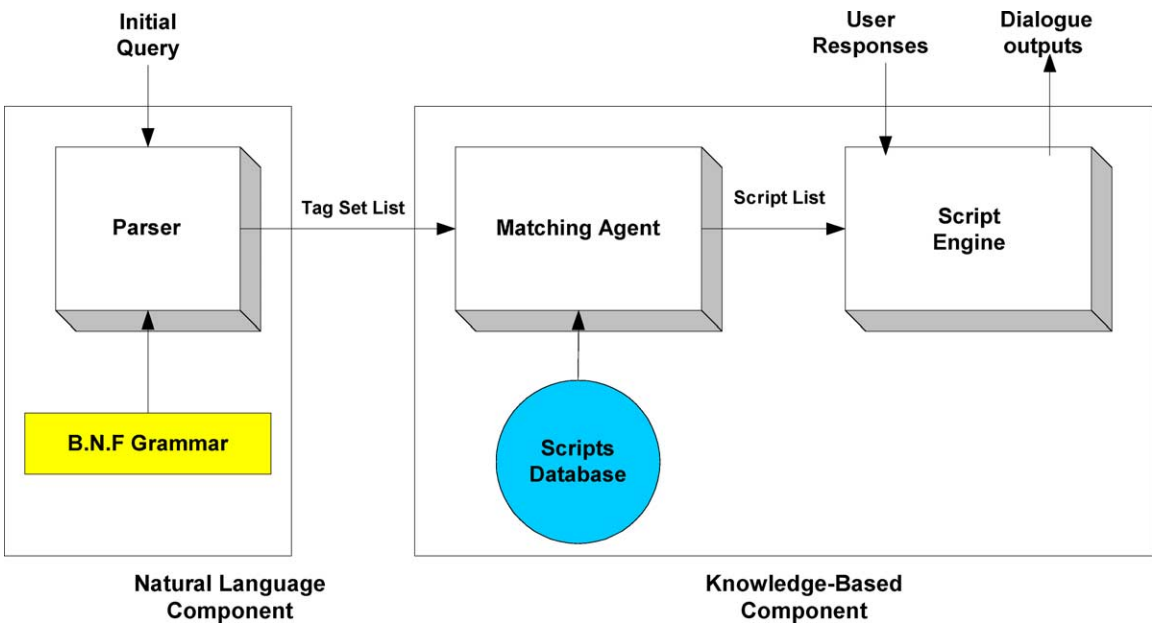

Fig. 1. Components of the mixed initiative dialogue system.

information extraction systems and extends of Plant's work on knowledge-based help systems [12].

2. Language choice

A major issue to decide upon is how to represent the BNF syntax as a data structure. However, a pre-requisite factor on the nature and form of the data structure is the language with which the whole system is to be implemented. There is a wide choice of languages with which the implementation could be performed:

- Procedural (imperative) languages such as Pascal [26,27], C [28], Perl [29]
- Specialist languages for NLP such as LIFER [10].
- Assumptive Logic Programming [30],
- Object languages such as Visual Basic [31], Object-orientated Languages such as C++ [32], Java [33] and Smalltalk [34]
- Traditional artificial intelligence languages of Prolog [35,36] (Clive, 1998) and Lisp [37,38]. Both Practical Prolog and Common Lisp can be considered as imperative in nature as they utilize constructs that possess side effect characteristics. Practical uses of Prolog frequently utilize features like cuts, side-effects, and database manipulating predicates [39] and Common Lisp utilizes such features as set expressions and format expressions, where as 'pure' Lisp such as Lispkit [40] does not permit constructs with side effects and is considered a functional language.
- Modern Functional languages have built upon pure Lisp's foundations and led to the development of languages such as Kent Recursive Calculator (Turner, 1982) Miranda [41], and Haskell [42].

The functional language AFL[1] was chosen for many reasons; however the main advantage this class of language has over traditional procedural languages is that it effectively reduces the amount of work the programmer has to perform. It does this in two ways: first, by handling allocation of storage. The second, which is more important, is that the system assumes all responsibility for the evaluation order of the functions, removing the problems associated with structuring the program in order to obtain the desired sequence of evaluation.

The 'lazy evaluation' strategy (Turner, 1982) used in this class of language also has the advantage of greatly simplifying the parsing process. In order to perform the task of parsing ambiguous grammars, it is essential for the system undertaking the task, to use a form of lazy evaluation in order to search for all possible solutions. The logic based languages such as Prolog use the technique of backtracking to do this, however, this is not as easy to use as lazy evaluation. Implementation in AFL also saves the programmer from having to explicitly write any backtracking code.

The use of infinite (or more precisely, unbounded lazily constructed) data structures is also of significant advantage in that they are ideal for processing the input and output of a parsing program that can be regarded as infinite streams of information. This approach to input/output, by-passes the problem of explicit 'reads' and having to decide upon

[1] AFL is a purely functional programming language, an extension of KRC developed by Stephen Murrell and Alessandro Warth. The system together with the programs described in this paper ccan be downloaded from http://rabbit.eng.miami.edu/afl/download/helper.html.

the sequencing of all events prior to processing, also by programming in terms of infinite I/O streams this allows program modules to be combined easily.

Pattern matching is a feature of functional programming languages that is extremely useful in the domain of NLP. These techniques allow complex conditions to be expressed very simply, especially when the functions have many arguments, reducing the need to use guards.

The functional approach also allows the use of higher order functions and a recursive equation style of programming. These powerful features combined with the use of set expressions (Turner, 1982) allow for more readable, shorter programs to be developed.

3. A functional specification of BNF

Having decided upon the BNF form of grammar and the use of the functional language AFL with which to ultimately implement the parser, the next stage was to devise a representation of the BNF in AFL.

There are four components making up the BNF: (i) Terminal symbols, e.g. words like 'dog', (ii) Non terminal symbols, e.g. <noun> these being the name of structural units and denoted by enclosure within angle-brackets, (iii) The disjunction of two or more components, the 'or' being represented by the '|' symbol, and (iv) The conjunction of two or more components with the 'and' being represented by juxtaposition. We extend this with a fifth component denoting a semantic tag.

For example,

$$< \text{s} > \ :: = < \text{noun} > | < \ \text{verb} >$$

means <s> is either a <noun> or a <verb>, whereas

$$< \text{s} > \ :: = < \text{noun} > \ < \text{verb} >$$

means <s> must consist of a <noun> followed by a <verb>

It was decided to represent these components in terms of lists and by defining some reserved words to be recognized by the parser.

Translating BNF to AFL, terminal symbols remain in a similar format; the word **dog** becomes the string `"dog"`. Non-terminal symbols now become single element lists, e.g. <noun> would become `["noun"]`. The reserved words 'or' and 'seq' were then introduced, allowing the conjunction and disjunction of symbols. For example,

$$< \text{verb} > | < \text{noun} >$$

is represented as

```
["or", ["verb"], ["noun"]]
```

and the sequence

$$< \text{verb} > < \text{noun} >$$

by

```
["seq", ["verb"], ["noun"]]
```

Semantic tags are introduced by the reserved word 'tag'. For example:

```
["tag", ["seq", ["verb"], ["noun"]],
"imperative"]
```

means that `["seq", ["verb"], ["noun"]]` should be parsed and a semantic tag added noting the fact that this sentence is 'imperative'.

Whole BNF descriptions are collections of named rules. In AFL BNF is represented as a structured list, the first item in each sublist is the name of the rule and the rest of each list is the definition.

$$< \text{s} > \ :: = < \text{noun} > < \text{verb} >$$
$$< \text{noun} > \ :: = \text{“cat”} | \text{“dog”}$$

would be represented by the definition:

```
bnf = = [["s",    ["seq",    ["noun"],
["verb"]]],
["noun", ["or", "cat", "dog"]]]
```

Having devised a representation in which to express these components we were are able to write equations for any set of BNF definitions.

Thus, the BNF for a very simple subset of English, just sufficient for these examples can be expressed in the functional notation:

```
[ ["s", ["seq", ["nphr"], ["verb"], ["rest"]]],
  ["nphr", ["seq", ["artic"], ["adjs"], ["nouns"]]],
  ["artic", ["or", "the", "a", "an", "some"]],
  ["adjs", ["or", ["adj"], ["seq", ["adj"], ["adjs"]]]],
  ["adj", ["or", ["big", "red", "quick", "brown", "lazy"]]],
  ["nouns", ["or", ["noun"], ["seq", ["noun"], ["nouns"]]]],
  ["noun", ["or", "fruit", "fly", "flies", "banana", "fox", "dog", "dogs"]],
  ["verb", ["or", "jump", "jumps", "fly", "flies", "like", "likes"]],
  ["rest", ["or",  ["object"], ["descr"]]],
  ["object", ["nphr"]],
  ["descr", ["seq", ["or", "like", "as"], ["nphr"]]] ]
```

And the sentence 'fruit flies like a banana' can be expressed as a list of terminal symbols:

```
Sentence = ["fruit", "flies", "like", "a",
"banana"]
```

Of course for the UNIX help domain the grammar is larger and more complex but follows the same pattern, it is included as Appendix B.

4. The parser

Having devised the BNF specification and its functional representation the next step was to develop a parser which would accept input in the form of a sentence in English and the BNF for the grammar, check to see if the sentence is legal according to the grammar, parse it and extract semantic information from it.

The utilization of a parser-based system allows the system to extract meaning from the interaction and not just attempt to match key words at random. It is important to note the advantages of this approach over the much simpler (Eliza-Like) keyword scanning or pattern directing systems. The word 'file' could represent the action of placing a document in a data store ('file this under expenses') or it could refer to a simple disk file object: two completely different meanings. In this system, actually parsing the input provides a context for each word—when 'file' is encountered, we know whether it is in a verb or a noun context, and can easily provide the appropriate specific semantic tag.

The following example illustrates the advantage of parser-based systems over pure pattern matching keyword-based systems. If a user new to the UNIX operating system needed for some reason to search a file for a pattern, then under keyword pattern matching this can cause some problems. Firstly, the user being a novice does not know the commands mnemonic name, so retrieval of the information is not available through that approach. Secondly, if the keyword 'search' was tried and no solution found, the next step for the user would, perhaps be, to search under 'file,' but this through the keyword lookup facility of UNIX 'apropos file,' produces five screens of possible commands associated with files. The only way to find the desired 'grep' command would be to divide and conquer the situation. Thus, extracting meaning from a piece of text is vital to efficient and effective problem solving.

The first stage in the creation of the parser involves allowing the user to input an English sentence in free form. The system then takes the input and translates it into a list of words.

Thus, the sentence input as:

fruit flies like a banana

becomes:

```
["fruit", "flies", "like", "a", "banana"]
```

Having achieved this, the next and most important stage was to devise the functions to parse the list of words. The most important function upon which the recursive nature of the parser is based is 'match'.

Match is a function whose type ideally would be:

$$\text{match: BNF} \times \text{LIST(WORDS)} \rightarrow \text{PARSE\_TREE}$$

It takes two arguments: A fragment of BNF and a list of words, which it then attempts to match together. In the ideal case the function having performed its matching operation would return a single parse tree. This however can only occur when the grammar produces nothing but unambiguous parses.

Due to the problem of ambiguous grammars the function Match has the type:

$$\text{match: BNF} \times \text{LIST(WORDS)} \rightarrow \text{P(PARSE\_TREE} \times \text{LIST (WORDS))}$$

Meaning that the output from match is a set of pairs representing all valid parses. The first element of the pair being the parse tree and the second being the list of, as yet, unused words that remain after the function match done its job with to match the input list of words and BNF fragment. If the match does not succeed then an empty set results.

In the case of an unambiguous parse, match returns a set containing just one pair where the first element is the only valid parse and the second element is the remaining input. When the whole sentence is matched unambiguously again a set containing only one pair is returned, this being the whole parse tree and the remaining input which should be empty as all of the sentence has been parsed.

Thus,

(i) Failure results in the empty set { }
(ii) An unambiguous parse in {<the only possible parse >, <remaining input > }
(iii) An unambiguous parse of the whole sentence gives: {<the only possible parse tree >, < > }

The function match has to be able to cope with any fragment of the BNF that it is given. This can be broken down into four major cases:

(i) Terminal symbols, e.g. 'dog'
(ii) Non-terminal symbols, e.g. ["noun"]
(iii) A list of alternatives, e.g. ["or", alt^1, alt^2, …, alt^n]
(iv) A sequence of parts, e.g. ["seq", $part^1$, $part^2$, …, $part^n$]

When match encounters a terminal symbol it takes the symbol and the list of words and tries to match them. If the symbol matches (i.e. is equal to) the head of the list of words then this is a unique parse and the result will be a set containing one pair as in the description above. This pair consists of the symbol and the remainder of the list of input words. If the symbol does not match the empty set if returned indicating failure.

Hence, the equations of match that covers the terminal symbols are:

```
match word (word:inp) = [[[word], inp]]
match word (other:inp) = [ ]
```

The case of non-terminal symbols can now be examined. If the match function receives as its first argument a non-terminal symbol then an attempt is made to expand it into the corresponding fragment of BNF. This being done through a function called 'lookup', which takes as its arguments the symbol that it is trying to expand and the whole of the BNF definitions; it then searches the left hand sides of the BNF productions for a match, if it is successful the expanded definition is returned, this expanded definition is given along with the original list of words to another call of match.

The equation that covers non-terminal symbols therefore is:

```
match [nts] inp = name (match (lookup
nts bnf) inp) nts
```

For example when the root of the BNF; $<$s$>$, is given to match along with a list of words:

```
match ["s"] ["list", "of", "words"]
```

the lookup function would transform ["s"] into:

```
["seq", ["noun"], ["verb"]]
```

and the equation above would then call match again, this time with:

```
match ["seq", ["noun"], ["verb"]]
["list", "of", "words"]
```

If the fragment of BNF to be matched against the list of words was composed of alternatives any of which could be matched then a separate case is needed. In this case, each of the alternatives is matched against the same list of words, each match resulting in a set of possible parses. For example, if the list of words to be matched was ["a", "b", "c"] and the BNF is defined by:

```
["or", ["seq", "a", ["seq", "b", "c"]],
       ["seq", ["seq", "a", "b"], "c"],
       ["seq", "c", "d", "e"]]
```

Then the resulting sets would be

```
{[["a", "b", "c"], [ ]]}
{[["a", "b", "c"], [ ]]}
{ }
```

The union of which clearly gives all the valid parses for the whole construct:

```
{[["a", "b", "c"], [ ]]}
```

So the union operation is simulated by appending all of the resulting sets together giving:

```
[[["a", "b", "c"], [ ]], [["a", "b",
"c"], [ ]]]
```

The append operation can result in the repetition of pairs within the list (as above); of course it would be possible to write a function to look for and remove repetitions, however this is unnecessary because the appearance of a parse more than once in the list does not cause concern as it is treated as another valid ambiguous parse.

The matching of alternatives is handled by the equations:

```
match ("or":prodlist) inp = matchor
prodlist inp
matchor [ ] inp = [ ]
matchor (item: prodlist) inp = match item
inp++
matchor prodlist inp
```

If the fragment of BNF to be matched is a sequence, then match has to try to match this sequence with the list of words. The initial attempt is a match between the first part of the sequence and the first portion of the words in the list. If this match is successful then the parser will attempt to match the rest of the sequence with the rest of the words. In order for the rest of the sequence to be matched it is necessary to know which words remain unparsed. This is why successful matches return a remaining word list along with each parse tree. When a successful match for the rest of the sequence is found, the new parse tree and the previous one are combined to form a single tree. Match thus produces a new list of pairs and by doing this for all parts in the sequence, producing all possible parses for the whole sequence.

The functions that handle sequences being:

```
match ("seq":prodlist) inp = matchseq
prodlist inp
matchseq [ ] inp = [[[ ], inp]]
matchseq (item:prodlist) inp =
matchseq' (match item inp) prodlist
matchseq' [ ] prodlist = [ ]
matchseq' ([tree,inp]:rest) prodlist =
```

```
combine tree (matchseq prodlist inp)++
matchseq' rest prodlist
```

'combine' is the function that combines trees together. It takes the tree produced by a new application of match and appends it onto the end of the previous version of the tree, building a new tree from the two subtrees. 'combine' is defined by:

```
combine t1 [ ] = [ ]
combine  t1  ([t2,  i2]:rest) = [t1++
t2,i2]:(combine t1 rest)
```

This covers the fundamental elements of BNF. There are other clauses to the definition of match, which cover the processing of semantic tags and the preservation of the syntactic structure, for details the reader is referred to Appendices A and B, which contain the complete parsing and processing program, and the grammar.

As it is currently implemented, the system has its entire vocabulary built into the grammar (for example, there is a production in the grammar saying $< place > ::= africa|atlanta|canada|\ldots$, see Appendix B). For a complete system this would be most unsatisfactory, but it is not a difficult matter to make the system actively look up words in a database instead. In fact, the *WordNet* system [43] provides an ideal source for this information: it is an extensive list of English words and proper nouns, complete with semantic tags and other information, in a fairly easy to interface format.

In reality, the processing of the input is a little more complicated than suggested above. It would be very inconvenient to leave semantic tags in the parse tree, and require later stages of the program to scan for them. Instead, the parser keeps the tags separately, and creates lists of *three* items: parse tree, remaining input words, and tag set. Of these three items, only the tag set is ultimately useful. When parsing is complete, only the list of tag sets from successful parses is kept. Due to ambiguity in natural languages, there may be a number of different tag sets; they are all provided in a list, to the knowledge base processor.

5. The knowledge base processor

Having constructed the parser and a functional specification of the BNF grammar, the next stage was to make the system respond to the user queries with meaningful answers. This was achieved through the use of a knowledge base that used as its representational structure an association list.

The association list connects pre-constructed scripts and informative texts to patterns or templates for semantic tag lists. For example, a user wanting to know how to print a file may type 'I want to print a file', which would be reduced by the parser to the tag set `[["action", "print"], ["object", "datafile"]]`. Information about how to print files may have as its index in the association list `[["action", "print"], ["object", "*"]]`, indicating that it would be a relevant response to a question about printing something. All of the possible tag sets are compared against all of the possible association list indexes, and a score is computed for closeness of match. The entry with the highest score is selected for presentation to the user as the answer. In a more sophisticated version, it would of course be possible to offer the user a 'menu' of the highest scoring entries to select from. The knowledge base appears in Appendix C.

This form of knowledge base provides a means of outputting textual answers to direct questions. An example of the kind of dialogue that can be achieved is:

```
this is the UNIX help system—how can I
help you?
  ctrl-D to exit, then:exit to leave
AFL)
  I want to print the budget
  best score = 100: print-something…
  If you want to print a file, the 'lpr'
command is what you need. If what you
want to print isn't in a file yet, you need
to get it in a file first; see the
documentation for the application you
are using. If you simply want to print
output that would normally appear on
your terminal, type the usual command,
followed by '| lpr' on the same line, e.g.
cal 2003 | lpr
```

Thus, it can be seen that even from this limited BNF grammar it is possible to answer a variety of query types ranging from questions on specific topics to general queries. However, it became apparent that another format of query that had to be catered for was the input of abbreviated queries. For example, one of the first things that people try when experimenting with a help system is to input the mnemonic of the command under investigation (this may be due to the lack of adequate help systems in the past [13]. Thus, interactions such as the following were developed:

```
this is the UNIX help system—how can I
help you?
  ctrl-D to exit, then:exit to leave
AFL)
  cal
  best score = 100: cal-command…
  The cal command produces a calendar
for any given month or year. It takes
as parameters: (optionally) the month
[1-12], followed by the year [all
digits]. It can also work out the date of
Easter.
```

A one-sided dialogue of this type could produce responses no better than the existing Unix help systems; one word of input does not provide enough information for anything better. An improvement on this basis, is the provision of a 'mixed initiative' dialogue capability to provide the user with a more stimulating interaction, allowing far more probing questions to be asked and more meaningful answers given.

6. Towards a mixed initiative dialogue

In order to build a mixed initiative framework it was necessary to extend the knowledge base. However, it was felt that the specialized knowledge required for a dialogue should be separated from the general information and facts stored in the knowledge base [44].

One of the underlying aims of developing this functional online help system was to utilize the functional programming systems formality. This powerful aspect of the language was beneficial in several ways, for example the heterogeneous set of system components could be integrated together, and the knowledge base could be modularized in a fashion that was felt to be beneficial from the perspective of both verification and validation [45]. Utilizing this factor, it was decided that the use of script based processing would be an applicable technique to employ. The original concept behind scripts was that they specify the normal or default sequence of events; as well as exceptions and possible errors, associated with a particular situation.

Recent research, developed from this area has focused upon the idea of 'Information Extraction,' (IE) which has been defined as 'a process aiming at combining, within a priori defined structures, data extracted out of texts. The result of the combinations may correspond to information contained either implicitly or explicitly in texts' [46]. The interesting aspect of IE versus traditional 'Information Retrieval' (IR) is that within IE the focus is upon the 'structure of texts' [47] rather than the IR perspective where 'texts are just bags of words' [47]. The ability to extract information is based upon the structures used within texts, more specifically template structures. The methodology in this paper builds upon and modifies Hobbs [48] generic IE System parameters as this research develops an online help system through a functional equations style of programming and uses the dialogue to develop the scripting technique rather than extracting data from other data sources, although this is a possible future extension to the project.

In our prototypes' domain, the UNIX operating system, users frequently have problems memorizing the parameters associated with a given command. Thus, this was felt to be a suitable area with which to experiment with the use of knowledge-based scripting. The parser again needed to be modified in order to enable the most appropriate script selection to be performed. The selection process take into account the current situation and context upon which the query is based. In order to decide which script is the most appropriate the information list produced by the parser is utilized. The script control system matching the information list with the association list. If the attempt is successful the control of the dialogue is passed on to the script. However, if no existing script is appropriate for that query the system prompts the user for the next query.

The structure content and use of a script is best explained through the use of an example:

If a user were to ask the question:

how can I see a calendar?

Based on the semantic tags, the system selects two objects: a non-interactive knowledge based article (perhaps just a 'man page') for display as the primary response, and an interactive script that the user may then elect to run through.

The system includes a scripting language so that users unfamiliar with programming can easily set up scripts to explain any situation. We do not expand upon the exact details of the scripting language here, the beginning of the script for the lpr command is shown below, and the whole interpreter with two complete scripts can be seen in Appendices D and E.

```
lprscript == [ \
  ["start", \
     "Would you like to run through a use of the 'lpr' command? ", \
     "readstr", @ans, \
     "if", _@ ans~"no" || ans~"n" @_, "end", "else", "goto", "where" ], \
  ["where", \
     "Which printer do you want the file to appear on:\n", \
     "    (1)  The fast laser printer in the sewing room,\n", \
     "    (2)  The colour laser printer by the ironing board,\n", \
     "    (3)  The old dot-matrix in Bunty's room,\n", \
     "    (4)  The slow laser printer in the dolphinarium.\n", \
     "  enter printer choice (1-4): ", \
     "readnum", @which, \
     "if", _@ which=1 @_, "set", @printer, "-Plp ", "goto", "email", "else", \
     "if", _@ which=2 @_, "set", @printer, "-Pclr ", "goto", "email", "else", \
     "if", _@ which=3 @_, "set", @printer, "", "goto", "email", "else", \
     "if", _@ which=4 @_, "set", @printer, "-Porpoise ", "goto", "email", "else", \
     "you have to answer 1, 2, 3, or 4\n", \
```

The use of scripts provides a means by which users can do more than just access a help system that gives them textual information in a standard form leaving the user to decipher its often-cryptic content. Its intention is to animate the manual and for the system to act as an advisor rather than a reference. This is illustrated through the following example, where the system guides the user through the intricacies of the sort command:

```
this is the UNIX help system—how can I help you?
  ctrl-D to exit, then:exit to leave AFL)
  how can I see a calendar?
  best score = 200: view-calendar…
  The cal command produces a calendar for any given month or year. It takes as parameters: (optionally) the month [1-12], followed by the year [all digits]. It can also work out the date of Easter.
  Would you like to run through a use of the 'cal' command? Yes
  Do you want a calendar for a whole year (Y) or just a month (M)
  or do you just want to be told when easter is (E)?
  (enter 'Y', 'M', or 'E'): M
  Which year do you want to know about? 2003
  Which month do you want the calendar for? August
  a month must be between 1 and 12,
  Which month do you want the calendar for? 8
  The command to type is 'cal 8 2003'
```

From this simple example we can see the potential of the scripting system to provide a very useful extension to the knowledge-based component, filling the gap between the standard help facility and the human 'help desk' advisor. The utilization of a scripting template approach follows from the premise that they 'provide a method for organizing large amounts of knowledge needed to perform cognitive tasks' [49]. The functional approach to the construction of the system presents several advantages that were described at the outset of the paper, including: Brevity, clarity, polymorphism, lazy evaluation, encapsulated abstraction and memory management (http://www.haskell.org/) [57] which facilitate the scalability of the approach and the ability of functional language be embedded in script slots, facilitates the potential extension of this feature to provide 'active scripts' that fire dynamic functions as required. The functional style of programming is an active research area and issues surrounding the run time performance of compiled functional programming systems have been under investigation [50,51] as have graphical user interface run time environments that significantly aid system development [22].

7. Future work

Research in this area can be extended in both the development of the theory of NLP and in the development of functional programming as applied to the area of NLP. The two areas could be extended along the lines suggested by Hobbs (1995) and allow for the automatic generation of the functional equations of specified grammars, together with the automatic generation of scripts and templates from databanks as suggested in the emerging area of information extraction [47]. Adaptive systems that learn the behavior of their user group would also be an area for study.

Developing functional programming systems to meet these challenges will benefit from the research in compiled functional programming as well as benefit from the utilization of a graphical interface facility within a functional programming environment as developed by Addis [22].

Appendix A. Parser and controlling program

The material of the appendices, together with the software required to run the system under unix, can be downloaded from http://rabbit.eng.miami.edu/afl/download/helper.html

```
:print

# clean x:  [[1,2,3],[4,[],6],[7,8,9],[10,[],12]] -> [[4,6],[10,12]]
#    reduces list of parses to list of successfull tree x tag-set pairs
clean [] == []
clean ([[tree], ["*END*"], info]:rest) == [tree, (reverse info)]:clean rest
clean ([[tree], inp, info]:rest) == clean rest

# name parselist n: puts label n on root or all trees in list
name [] nts == []
name ([tree, inp,  info]:rest) nts == [[nts: tree], inp, info]:name rest nts
```

```
# match productionrule input bnf note -> list of [tree, remaining-input, tag-set]
match [] inp info note == [[[], inp, info]]
match ["none"] inp info note == [[[], inp, info]]
match patt [] info note == []
match [nts] inp info note == name (match (lookup nts bnf []) inp info note) nts
match ("or": prod) inp info note == matchor prod inp info note
match ("seq": prod) inp info note == matchseq prod inp info note
match ["tag", prod, tag] inp info note == matchtag prod tag inp info note
match ["note", prod, nn] inp info note == match prod inp info nn
match ["irrelevant", prod] inp info note == match prod inp info 0
match ["opt", prod] inp info note == matchopt prod inp info note
match pword (aword:inp) info note == [[[["word", pword]], inp, info]] <| pword~aword |> []

# matchor: deals with choices for match
matchor [] inp info note == []
matchor (prod:prodl) inp info note == match prod inp info note ++ matchor prodl inp info note

# matchopt: deals with optional things
matchopt prod inp info note == match prod inp info note ++ [[[], inp, info]]

# matchtag: deals with tags for match
matchtag prod tag inp info 0 == [] <| match prod inp info 0 = [] |> match prod inp info []
matchtag prod tag inp info [] == [] <| match prod inp info [] = [] |> match prod inp ([tag]:info) []
matchtag prod tag inp info note == [] <| match prod inp info note = [] |> match prod inp ([note,tag]:info) note

# matchseq and combine: deals with sequences for match
matchseq [] inp info note == [[[], inp, info]]
matchseq (prod:prodl) inp info note == matchseqprime (match prod inp info note) prodl note
matchseqprime [] prodl note == []
matchseqprime ([tree, inp, info]:rest) prodl note == combine tree (matchseq prodl inp info note) \
                                                     ++ matchseqprime rest prodl note
combine t1 [] == []
combine t1 ([t2, i2, info]:rest) == [t1 ++ t2, i2, info]: combine t1 rest

# parse: input-word-list -> list of [tree, tag-set]
parse pos x == clean (match pos x [] [])

:load grammar

heading == ["\nthis is the UNIX help system - how can I help you?\n", \
            "(ctrl-D to exit, then :exit to leave AFL)\n\n"]

# searchkb takes tagset list, finds [score, kbase, script] with best score for any one tagset
searchkb tagsetlist == findbest tagsetlist 0 []
findbest [] bsc bsf == bsf
findbest (tagset:rest) bsc bsf == (findbest rest (nsc<|nsc>bsc|>bsc) (newa<|nsc>bsc|>bsf) \
                                   where { nsc == car newa }) \
                                  where { newa == findbest2 tagset kbase 0 [] }

# findbest2: takes single tagset and finds [score, kbase, script] with best score
findbest2 tagset [] bsc bsf == bsc:bsf
findbest2 tagset ((act:res):rest) bsc bsf == findbest2 tagset rest (nsc<|nsc>bsc|>bsc) (res<|nsc>bsc|>bsf) \
                                             where { nsc == score tagset act }

# score and appear: calculate score for a patternlist against an actual taglist
score [] actl == 0
score (pat:l) actl == (score2 pat actl) + (score l actl)
score2 pat [] == 0
score2 pat (act:l) == (score2 pat l) + (onescore pat act)

# onescore pattern tag: score for how well one tag matches a pattern
onescore [x,y] [x,y] == 100
onescore [x] [x] == 100
onescore [x,"*"] [x,y] == 50
onescore ["*",y] [x,y] == 20
onescore [x,y] [x,z] == -75
onescore pat act == 0

# showscores takes list of tagsets and shows all possible scores for kbase items
showscores [] == []
showscores (tagset:rest) == "Scores for " : show tagset : "\n" : showscores1 tagset kbase : showscores rest
showscores1 tagset [] == []
showscores1 tagset ((kbkey:kbitem):kbrest) == showscores2 (score kbkey tagset) kbitem : showscores1 tagset kbrest
showscores2 scr (name:rest) == "  " : scr : ": " : name : "\n"

# advise listof(tagset) and listof(listof words)
#   if no tagsets, admit you don't know
#   take best match for tagset and explain it
#       searchkb returns pair: kb-text-entry and script-function
advise tslist restofinput == "\n" : explain (searchkb tslist) restofinput

# explain tagset and listof(listof words)
#   say what the knowledge base contains (non-interactive)
#   then follow the script (interactive)
```

```
explain (0: stuff) restofinput == "Sorry, I haven't a clue.\n\n": startagain restofinput
explain [scr, name, kbinfo, script] restofinput == "best score=" : scr : ": " : name : "...\n\n" : \
                                                   kbinfo : "\n" : doscript script scripts restofinput
explain [scr, name, kbinfo] restofinput == "best score=" : scr : ": " : name : "\n\n" : \
                                           kbinfo : startagain restofinput
explain [scr, name] restofinput == "best score=" : scr : ": " : name : "\n\n" : \
                                   "No detailed information available.\n\n" : startagain restofinput
explain x restofinput == "Sorry, I haven't a clue.\n\n": startagain restofinput

doscript sn [] in == startagain in
doscript sn slist in == obeyscript sn slist in

# loop: list of (list of words) -> the output
#   for each line or list of words:
#      parse and reduce to list of [parsetree, tagset]
#      reduce that to a list of tag sets
#      advise based on tagsets
loop [] == ["end\n\n(\":exit\" to exit AFL)\n\n"]
loop (exp:rest) == advise (map cadr (parse ["s"] exp)) rest
startagain restofinput == heading: loop restofinput

prettyprint x == prettyprint1 0 x
prettyprint1 ind [] == []
prettyprint1 ind ["word",x] == prettyprint2 (ind+3) : "WORD: \"" : x : "\""
prettyprint1 ind [a] == (prettyprint2 (ind+6) : a) <| isstring a |> (prettyprint1 (ind+3) a)
prettyprint1 ind (a:x) == prettyprint1 (ind+3) a : "\n" : prettyprint1 ind x
prettyprint1 ind x == prettyprint2 ind : x
prettyprint2 0 == []
prettyprint2 n == " " : prettyprint2 (n-1)

delimit [a,b] == ["SYNTAX", a, "SEMANTICS", b]
delimit a == ["UNEXPECTED", a]

# help: main function, parameter should be INPUTCHARS
#   divide list of input characters into lines
#   convert each line of chars into list of words
#   loop (i.e. process) that list of (list of words)
help in == heading: loop (map charstowords (spliton '\n' in))

analyse inch == enumerate prettyprint (map delimit (uniques (parse ["s"] (car (map charstowords (spliton '\n'
inch))))))

testparse inch pos == enumerate prettyprint (map car (parse [pos] (car (map charstowords (spliton '\n' inch)))))

justparse inch == testparse inch "s"

testscores inch == showscores (map cadr (parse ["s"] (car (map charstowords (spliton '\n' inch)))))

:load kbase

scripts == []

:load scripts

"\n\nType 'help inputchars' to run the whole thing,"
"or 'analyse inputchars' to see parse trees and semantic tags,"
"or 'testscores inputchars' to see scores of different kbase entries,"
"or 'justparse inputchars' to see parse trees only,"
"or 'testparse inputchars \"command\"' (e.g.) to try parsing as a command.\n"
```

Appendix B. Grammar

```
bnf == \
 [["articly", ["or", ["article"], ["posesiv"], ["demonst"]]], \
  ["article", ["or", "the", "a", "an", "some"]], \
  ["testone", ["seq", ["article"], ["adjctiv"]]], \
  ["demonst", ["or", "this", "that", "these", "those"]], \
  ["nompron", ["or", ["tag", ["or", "i", "you", "he", "she", "we", "they"], "person"], "it"]], \
  ["accpron", ["or", ["tag", ["or", "me", "you", "him", "her", "us", "them"], "person"], "it"]], \
  ["posesiv", ["or", "my", "your", "his", "her", "its", "our", "their", \
                     "someones", "anyones", "noones", "somebodys", "anybodys", "nobodys"]], \
  ["simpsub", ["or", ["nompron"], ["noungrp"]]], \
  ["subject", ["or", ["seq", ["descrip"], ["simpsub"]], \
                     ["person"], \
                     ["place"]]], \
  ["simpobj", ["or", ["accpron"], ["noungrp"]]], \
  ["sobject", ["or", ["seq", ["descrip"], ["simpobj"]], \
                     ["person"], \
                     ["place"], \
                     ["seq", ["opt", ["articly"]], ["amethod"]]]], \
```

```
 ["descrip", ["irrelevant", ["seq", ["opt", ["articly"]], ["opt", ["adjects"]]]]], \
 ["adjects", ["seq", ["opt", ["modiadj"]], ["adjctiv"], ["opt", ["adjects"]]]], \
 ["noungrp", ["seq", ["opt", ["onenoun"]], ["onenoun"]]], \
\
 ["amethod", ["seq", ["or", "how", "way", "method"], "to", ["command"]]], \
\
 ["modiadj", ["or", "very", "quite", "not", "highly"]], \
 ["adjctiv", ["or", ["sizeadj"], ["colradj"], ["badadj"], ["goodadj"], ["insltdj"], ["emotadj"], \
                    ["gnrladj"], ["techadj"], ["fulladj"]]], \
 ["sizeadj", ["or", "big", "small", "little", "tiny", "large", "fat", "thin", "tall", "short", "giant"]], \
 ["colradj", ["or", "black", "white", "grey", "gray", "red", "green", "blue", "yellow", "brown", "orange"]], \
 ["goodadj", ["or", "good", "nice", "new", "pretty", "clean"]], \
 ["fulladj", ["tag", ["or", "full", "empty", "filling", "stuffed"], "capacity"]], \
 ["badadj",  ["tag", ["or", "bad", "evil", "surly", "broken", "old", "smelly", "dirty", "damaged", "faulty", \
                           "hideous", "ugly", "slow", "noisy", "smoking"], "bad"]], \
 ["insltdj", ["tag", ["or", "stupid", "idiotic", "brainless", "moronic", "useless", "dim", \
                           "dopey", "stinking", "****ing"], "bad"]], \
 ["emotadj", ["or", "happy", "sad", "angry", "cross", "irate", "miserable", "grinning"]], \

 ["gnrladj", ["or", "useful", "heavy", "light", "dark", "striped", "speckled", "hot", "important", \
                    "lazy", "quick", "fast", "hairy", "bald"]], \
 ["techadj", ["or", "electric", "electronic", "laser", "ethernet", "lcd", "led", "magnetic", "optical"]], \
\
 ["person",  ["tag", ["or", "john", "mary", "bob", "bub", "robert", "alex", "carolyn", "jim", "jane", \
                           "june", "arthur", "brian", "jilly", "abigail", "grumbellina"], "person"]], \
 ["place",   ["tag", ["or", "africa", "atlanta", "canada", "kalamazoo", "portland", "hell", "brazil", \
                           "washington", ["seq","new","york"], "france", "sidcup"], "place"]], \
\
 ["onenoun", ["or", ["animann"], ["personn"], ["prfesnn"], ["afilenn"], ["hrdwrnn"], \
                    ["atoolnn"], ["afoodnn"], ["prgrmnn"], ["mediann"], ["resrcnn"], \
                    ["othernn"], ["unixcmd"], ["insctnn"]]], \
 ["insctnn", ["tag", ["or", "fly", "flies", "spider", "spiders", "ant", "ants", "insect", "insects", "bee", \
                           "bees", "bug", "bugs", "wasp", "wasps", "chihuahua"], "insect"]], \
 ["animann", ["tag", ["or", "cat", "cats", "dog", "dogs", "fox", "foxes", "bat", "bats", "tiger", \
                           "tigers", "lion", "lions", "badger", "badgers", "animal", "animals", "creature", \
                           "creatures", "horse", "horses", "octopus", "octopodes", "squid", "squids", \
                           "fish", "fishes", "monkey", "monkeys"], "animal"]], \
 ["personn", ["tag", ["or", "man", "men", "woman", "women", "lady", "ladies", "boy", "boys", "girl", "girls", \
                           "youth", "youths", "child", "children", "baby", "babies"], "person"]], \
 ["prfesnn", ["tag", ["or", "customer", "customers", "boss", "bosses", "manager", "managers", "user", \
                           "users", "programmer", "programmers", "president", "presidents", "lawyer", \
                           "lawyers", "student", "students", "editor", "editors"], "person"]], \
 ["afilenn", ["tag", ["or", "file", "files", "document", "documents", "spreadsheet", "spreadsheets", \
                           "database", "databases", "email", "emails", "data", "input", "output", "memo", \
                           "memos", "fax", "faxes"], "datafile"]], \
 ["hrdwrnn", ["tag", ["or", "computer", "computers", "cpu", "cpus", "memory", "chip", "chips", "processor", \
                           "processors", "machine", "machines", "printer", "printers", "scanner", \
                           "mouse", "mouses", "mice", "keyboard", "keyboards", "monitor", "monitors", \
                           "screen", "screens", "display", "displays", "block", "blocks"], "hardware"]], \
 ["atoolnn", ["tag", ["or", "hammer", "screwdriver", "knife", "fork", "spoon", "gun"], "tool"]], \
 ["afoodnn", ["tag", ["or", "food", "breakfast", "lunch", "dinner", "snack", "sausage", "sausages", "cake", \
                           "cakes", "chicken", "chickens", "beef", "pork", "bread", "butter", "tea", \
                           "coffee", "soda", "coke", "beer", "wine", "liquor", "pie", "pies", "sandwich", \
                           "sandwiches", "fruit", "banana", "bananas", "milk", "cheese"], "food"]], \
 ["prgrmnn", ["tag", ["or", "program", "programs", "application", "applications", "software", "process", \
                           "processes", "wordprocessor", "editor", "word", "excel"], "program"]], \
 ["mediann", ["tag", ["or", "disc", "discs", "disk", "disks", "floppy", "floppies", "cd", "cds"], "media"]], \
 ["resrcnn", ["tag", ["or", "quota", "quotas", "budget", "budgets", "limit", "limits", "allowance", \
                           "allowances", "maximum", "maximums", "maxima", "share", "shares"], "quota"]], \
 ["othernn", ["or", "nothing", "something", "here", "there", "home", "object", "thing", \
                           ["tag", "calendar", "calendar"]]], \
 ["personn", ["tag", ["or", "someone", "anyone", "noone", "somebody", "anybody", "nobody"], "person"]], \
 ["unixcmd", ["or", "command", ["tag", "cal", "cal-cmd"], \
                              ["tag", "sort", "sort-cmd"], \
                              ["tag", "lpr", "lpr-cmd"]]], \
\
 ["adverbs", ["seq", ["opt", ["modadvb"]], ["genadvb"]]], \
 ["modadvb", ["or", "very", "slightly", "not", "more", "less", "somewhat", "really", "quite"]], \
 ["genadvb", ["or", "quickly", "slowly", "cheaply", "happily", "sadly", "importantly", "politely", \
                    "really", "badly", "well"]], \
\
 ["verbbit", ["or", ["seq", ["opt", ["adverbs"]], ["oneverb"], ["opt", ["adverbs"]]]]], \
 ["oneverb", ["or", ["vbattac"], ["vbprint"], ["vbutlis"], ["vbdstry"], ["vbcnsum"], ["vbenjoy"], ["vbsave"], \
                    ["vbtravl"], ["vbacqir"], ["vbjumps"], ["verbdes"], ["vrbseem"], ["vbexist"]]], \
 ["vbattac", ["tag", ["or", "bite", "bit", "bites", "hit", "hits", "smash", "break", "broke"], "attack"]], \
 ["vbprint", ["tag", ["or", "print", "list", "publish", "type"], "print"]], \
 ["vbutlis", ["tag", ["or", "run", "execute", "use", "access", "see", "view", "display", "edit", "modify", \
                           "inspect", "change", "peruse", ["seq", "look", ["or", "over", "at"]]], "view"]], \
 ["vbdstry", ["tag", ["or", "delete", "deleted", "deletes", "kill", "killed", "kills", \
                           "erase", "erased", "destroy", "murdered", "remove"], "delete"]], \
```

```
 ["vbcnsum", ["tag", ["or", "consume", "eat", "eats", "ate", "drink", "drinks", "finish"], "consume"]], \
 ["vbenjoy", ["tag", ["or", "enjoy", "enjoys", "like", "likes", "love", "loves"], "enjoy"]], \
 ["vbtravl", ["tag", ["or", "fly", "flies", "walk", "walks", "drive", "drives", "run", "runs", "ride", \
                        "rides", "travel", "travels", "move", "moves", "go", "went"], "travel"]], \
 ["vbacqir", ["tag", ["or", "take", "takes", "took", "get", "gets", "got", "own", "buy", "acquire", "steal", \
                        ["seq", "pick", "up"], "grab"], "obtain"]], \
 ["vbjumps", ["tag", ["or", "jump", "jumps", "jumped", "leap", "leaps", "leapt", "leaped", "spring", \
                        "springs", "sprung", "sprang"], "jump"]], \
 ["verbdes", ["or", "want", "wants", "need", "needs", "require", ["seq", "must", "have"]]], \
 ["vrbseem", ["or", "look", "looks", "seem", "seems", "appear", "appears"]], \
 ["vbsave",  ["tag", ["or", "save", "preserve", "keep", "archive"], "save"]], \
 ["vbexist", ["tag", ["or", "is", "am", "are", "be", "exist", "become"], "is"]], \
 ["vbintro", ["or", "think", "thinks", "believe", "believes"]], \
\
 ["modifyr", ["or", ["modstyl"], ["modwith"], ["modprep"], ["modsent"]]], \
 ["modstyl", ["seq", ["or", "like", "as"], ["sobject"]]], \
 ["modwith", ["seq", "with", ["sobject"]]], \
 ["modprep", ["seq", ["preposi"], ["sobject"]]], \
 ["modsent", ["seq", ["or", "because", "for", "when", "like", "as"], ["statmnt"]]], \
\
 ["preposi",    ["or", "to", "from", "at", "over", "under", "near", "beside", "by", "on", "around"]], \
 ["shouldword", ["or", "can", "may", "should", "ought", "would", "do", "will", "did"]], \

\
 ["actionphrase", ["seq", ["note", ["verbbit"], "action"], \
                          ["opt", ["note", ["sobject"], "object"]], \
                          ["opt", ["note", ["modifyr"], "explanation"]], \
                          ["opt", ["adverbs"]]]], \
 ["bephrase",     ["seq", ["opt", ["adverbs"]], \
                          ["note", ["vbexist"], "action"], \
                          ["opt", ["adverbs"]], \
                          ["note", ["adjects"], "state"], \
                          ["opt", ["adverbs"]]]], \
 ["desirephrase", ["seq", ["opt", ["adverbs"]], \
                          ["verbdes"], \
                          "to", \
                          ["verbphrase"]]], \
 ["shouldphrase", ["seq", ["opt", ["adverbs"]], \
                          ["shouldword"], \
                          ["opt", ["adverbs"]], \
                          ["opt", "to"], \
                          ["verbphrase"]]], \
 ["beliefphrase", ["seq", ["opt", ["adverbs"]], \
                          ["vbintro"], \
                          ["opt", "that"], \
                          ["statement"]]], \
["verbphrase",   ["or", ["actionphrase"], ["desirephrase"], ["beliefphrase"], ["bephrase"]]], \
["statement",    ["or", ["seq", ["note", ["subject"], "actor"], \
                                ["or", ["actionphrase"], ["bephrase"], ["shouldphrase"]]], \
                        ["seq", ["irrelevant", ["subject"]], ["or", ["desirephrase"], ["beliefphrase"]]]]],\
["command",      ["verbphrase"]], \
["howquestion",  ["seq", "how", ["shouldword"], ["statement"]]], \
["whatquestion", ["seq", "what", ["irrelevant", ["vbexist"]], ["note", ["subject"], "explain"]]], \
["quickie",      ["seq", ["note", ["unixcmd"], "explain"], ["opt", ["or", ".", "?"]]]], \
["question",     ["or", ["howquestion"], ["whatquestion"]]], \
["s",        ["or", ["seq", ["statement"], ["opt", "."]], \
                    ["seq", ["question"], ["opt", "?"]], \
                    ["seq", ["quickie"], ["opt", "."]], \
                    ["seq", ["command"], ["opt", "."]]]]
```

Appendix C. Knowledge base

```
# entries in kbase list are lists of 2 to 4 items:
# item 1: tag-set to be matched
# item 2: (string) quick-reference name
# item 3: (optional, list of strings) knowledge base article
# item 4: (optional, string) name of script in scripts list

kbase == [ \
 [["action", "print"], ["object", "datafile"]], "print-a-file", [], "lprscript"], \
 [["action", "print"]], "print-something", printsomething, "lprscript"], \
 [["action", "delete"], ["object", "datafile"]], "delete-file"], \
 [["action", "delete"], ["object", "person"]], "commit-murder"], \
 [["action", "delete"], ["object", "insect"]], "squish-bug"], \
 [["action", "view"], ["object", "datafile"]], "view-file", [], "lprscript"], \
 [["action", "view"], ["object", "calendar"]], "view-calendar", calscript, "calscript"], \
 [["action", "view"], ["object", "person"]], "visit-person"], \
```

```
 [[["action", "view"], ["object", "animal"]], "visit-zoo"], \
 [[["action", "enjoy"]], "something-enjoys-something"], \
 [[["action", "travel"]], "something-travels"], \
 [[["action", "travel"], ["explanation", "*"]], "something-travels-in-some-way"], \
 [[["action", "save"], ["object", "person"]], "call-international-rescue"], \
 [[["action", "save"], ["object", "datafile"]], "save-file"], \
 [[["actor", "media"], ["state", "capacity"]], "disc-quota"]]


printsomething == \
[ "If you want to print a file, the 'lpr' command is \n", \
  "what you need. If what you want to print isn't in \n", \
  "a file yet, you need to get it in a file first; see\n", \
  "the documentation for the application you are using.\n", \
  "If you simply want to print output that would normally\n", \
  "appear on your terminal, type the usual command,\n", \
  "followed by '| lpr' on the same line, e.g. cal 2003 | lpr\n" ]

calscript == \
[ "The cal command produces a calendar for any given month \n", \
  "or year. It takes as parameters: (optionally) the \n", \
  "month [1-12], followed by the year [all digits]. It can also \n", \
  "work out the date of Easter.\n" ]
```

Appendix D. Scripting language

```
eval assl [op, a, b] == (function op) (eval assl a) (eval assl b)
eval assl [op, a] == (function op) (eval assl a)
eval assl [a] == eval assl a
eval assl a == a <| isnum a || isstring a |> \
               lookup a assl a <| isname a |> "NULL"

obeyscript sname slist in == ( "No script called " : sname : "\n": startagain in \
                                 <| scr=[] |> \
                               obey ["goto", "start"] scr [] in [] ) \
                                 where { scr==findpair sname slist }

obey ("goto":x:rest) s al in co == \
          ( "Undefined label " : x : " in script " : (car s) : "\n": startagain in \
              <| fnd=[] |> \
            obey (cdr fnd) s al in co ) \
              where { fnd == findpair x (cadr s) }
obey ("gosub":x:rest) s al in co == \
          ( "Undefined label " : x : " in script " : (car s) : "\n": startagain in \
              <| fnd=[] |> \
            obey (cdr fnd) s al in (rest:co) ) \
              where { fnd == findpair x (cadr s) }
obey ("follow":x:rest) s al in co == \
          ( "Undefined script " : x : " called from script " : (car s) : "\n": startagain in \
              <| newscr=[] |> \
            obey ["goto", "start"] newscr al in (rest:co)) \
              where { newscr == findpair x scripts }
obey ("end":rest) s al in co == obeyend s al in co
obey ("return":rest) s al in co == obeyend s al in co
obey ("halt":rest) s al in co == []
obey ("say":x:rest) s al in co == \
                (x <| isstring x |> eval al x) : obey rest s al in co
obey ("readline":x:rest) s al (in1:in) co == obey rest s ([x, in1]:al) in co
obey ("readline":x:rest) s al [] co == "\nInput Terminated\n" : startagain in
obey ("readstr":x:rest) s al ((in1:in2):in) co == obey rest s ([x, in1]:al) in co
obey ("readstr":x:rest) s al [] co == "\nInput Terminated\n" : startagain in
obey ("readnum":x:rest) s al ((in1:in2):in) co == obey rest s ([x, numval in1]:al) in co
obey ("readnum":x:rest) s al [] co == "\nInput Terminated\n" : startagain in
obey ("set":[@=,a,b]:rest) s al in co == obey rest s ([a,eval al b]:al) in co
obey ("set":a:b:rest) s al in co == obey rest s ([a,eval al b]:al) in co
obey ("if":x:rest) s al in co == (obey rest s al in co) <| eval al x |> \
                                 (obeyelse rest s al in co)
obey ("else":rest) s al in co == obeyend s al in co
obey (x:rest) s al in co == (x <| isstring x |> eval al x) : obey rest s al in co
obey [] s al in co == obeyend s al in co

obeyelse ("else":rest) s al in co == obey rest s al in co
obeyelse (x:rest) s al in co == obeyelse rest s al in co
obeyelse [] s al in co == obeyend s al in co

obeyend s al in [] == startagain in
obeyend s al in (co1:co) == obey co1 s al in co

scripts == [["lprscript", lprscript], ["calscript", calscript]]
```

Appendix E. Sample scripts

```
lprscript == [ \
  ["start", \
     "Would you like to run through a use of the 'lpr' command? ", \
     "readstr", @ans, \
     "if", _@ ans~"no" || ans~"n" @_, "end", "else", "goto", "where" ], \
  ["where", \
     "Which printer do you want the file to appear on:\n", \
     "    (1)  The fast laser printer in the sewing room,\n", \
     "    (2)  The colour laser printer by the ironing board,\n", \
     "    (3)  The old dot-matrix in Bunty's room,\n", \
     "    (4)  The slow laser printer in the dolphinarium.\n", \
     "  enter printer choice (1-4): ", \
     "readnum", @which, \
     "if", _@ which=1 @_, "set", @printer, "-Plp ", "goto", "email", "else", \
     "if", _@ which=2 @_, "set", @printer, "-Pclr ", "goto", "email", "else", \
     "if", _@ which=3 @_, "set", @printer, "", "goto", "email", "else", \
     "if", _@ which=4 @_, "set", @printer, "-Porpoise ", "goto", "email", "else", \
     "you have to answer 1, 2, 3, or 4\n", \
     "goto", "where" ], \
  ["email", \
     "Would you like to be notified by email when the printing is done? ", \
     "readstr", @yorn, \
     "if", _@ yorn~"y" || yorn~"yes" @_, "set", @email, "-m ", "goto", "delete", "else", \
     "if", _@ yorn~"n" || yorn~"no" @_, "set", @email, "", "goto", "delete", "else", \
     "you have to answer 'Y', 'yes', 'N', or 'no'\n", \
     "goto", "email" ], \
  ["delete", \
     "Is the file a temporary that you want deleted after printing? ", \
     "readstr", @yorn, \
     "if", _@ yorn~"y" || yorn~"yes" @_, "set", @del, "-r -s ", "goto", "last", "else", \
     "if", _@ yorn~"n" || yorn~"no" @_, "set", @del, "", "goto", "last", "else", \
     "you have to answer 'Y', 'yes', 'N', or 'no'\n", \
     "goto", "delete" ], \
  ["last", \
     "What is the name of the file you want to print? ", \
     "readstr", @fname, \
     "\nThe command is 'lpr ", @printer, @email, @del, @fname, "'\n", \
     "\nThen, to follow progress, use 'lpq ", @printer, "'\n" ] ]
calscript == [ \
  ["start", \
     "Would you like to run through a use of the 'cal' command? ", \
     "readstr", @ans, \
     "if", _@ ans~"no" || ans~"n" @_, "end", "else", "goto", "askyorm" ], \
  ["askyorm", \
     "Do you want a calendar for a whole year (Y) or just a month (M)\n", \
     "or do you just want to be told when easter is (E)?\n", \
     "(enter 'Y', 'M', or 'E'): ", \
     "readstr", @ans, \
     "if", _@ ans~"Y" @_, "goto", "wholeyear", "else", \
     "if", _@ ans~"M" @_, "goto", "onemonth", "else", \
     "if", _@ ans~"E" @_, "goto", "easter", "else", \
     "you have to answer 'Y', 'M', or 'E'\n", \
     "goto", "askyorm" ], \
  ["wholeyear", \
     "Would it be for this year or some other year? ('this' or 'other'): ", \
     "readstr", @ans, \
     "if", _@ ans~"this" @_, "goto", "thisyear", "else", \
     "if", _@ ans~"other" @_, "goto", "otheryear", "else", \
     "you have to answer either 'this' or 'other'\n", \
     "goto", "wholeyear" ], \
  ["thisyear", \
     "\nThe command to type is 'cal -y'\n", "end"], \
  ["otheryear", \
     "gosub", "getyear", \
     "\nTo get a calendar for ", @year, ", the command is 'cal ", @year, "'\n", \
     "end" ], \
  ["easter", \
     "Caluclating Easter, ", \
     "gosub", "getyear", \
     "\nTo find Easter ", @year, ", the command is 'ncal -e ", @year, "'\n", \
     "end" ], \
  ["onemonth", \
     "gosub", "getyear", \
     "gosub", "getmonth", \
     "\nThe command to type is 'cal ", @month, " ", @year, "'\n", \
     "end"], \
  ["getyear", \
     "Which year do you want to know about? ", \
```

```
   "readnum", @inyear, \
   "if", _@ inyear<=0 @_, "it must be a properly positive number,\n", \
                          "goto", "getyear", "else", \
   "if", _@ inyear<10 @_, "set", @year, _@ inyear+2000 @_, "else", \
   "if", _@ inyear<100 @_, "set", @year, _@ inyear+1900 @_, "else", \
   "set", @year, @inyear, \
   "return" ], \
["getmonth", \
   "Which month do you want the calendar for? ", \
   "readnum", @inmonth, \
   "if", _@ inmonth<=0 || inmonth>12 @_, \
         "a month must be between 1 and 12,\n", "goto", "getmonth", "else", \
   "set", @month, @inmonth, \
   "return" ]]
```

References

[1] M. Bannert, P. Reimann, Approaches to the design of software training, Journal of Computer Assisted Learning 16 (4) (2000) 281–284.

[2] A.F. Borthick, P.L. Bowen, D.R. Jones, M. Hung Kam Tse, The effects of information request ambiguity and construct incongruence on query development, Decision Support Systems 32 (1) (2001) 3–30.

[3] V. Owei, Natural language querying of databases: an information extraction approach in the conceptual query language, International Journal of Human Computer Studies 53 (4) (2000).

[4] J. Weizenbaum, A computer program for the study of natural language communication between man and machine, Communications of ACM 9 (1966) 36–45.

[5] J. Weizenbaum, Contextual understanding by computers, Communications of ACM 10 (1967) 327–360.

[6] N. Chomsky, Syntactic Sstructures, Mouton, The Hague, 1957.

[7] P. Postal, Limitations of phrase structured grammars, in: J.A. Fodor, J.J. Katz (Eds.), The Structure of Language, Prentice Hall, Englewood Cliffs, NJ, 1964, pp. 137–151.

[8] C. Filmore, The case for case, in: E. Bach, R. Harms (Eds.), Universals in Linguistic Theory, Reinhart & Winston, New York, 1968, pp. 1–88.

[9] M.A.K. Halliday, Categories of the theory of grammar, Word 17 (1961) 241–292.

[10] G.G. Hendrix, E.D. Sacerdoti, D. Sagalowick, J. Slocum, Developing a natural language interface to complex data, ACM Transaction Database System (1978) 105–147.

[11] J. Slocum, A Practical Comparison of Parsing Strategies, 1981, http://acl.ldc.upenn.edu/P/P81/p81-1001.pdf.

[12] R.T. Plant, An investigation of knowledge-based help facilities. MSc Dissertation. Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1985.

[13] G.M. Gwei, E. Foxley, Towards a consultative on-line help system, International Journal of Human–Computer Studies 32 (1990) 363–383.

[14] K. Sikel, How to compare the structure of parsing algorithms, in: G. Pighizzini, P. San Pietro (Eds.), Proceedings of ASMICS, Workshop on Parsing Theory, Milano, Italy, Oct 1994, 1994, pp. 21–39.

[15] J. Peterson, K. Mahesh, A. Goel, Situating natural language understanding within experience-based design, International Journal of Human Computer Studies. 1. 41 (6) (1994) 881–913.

[16] M. Mosny, Semantic information preprocessing for natural language interfaces to databases, 33rd Annual Meeting of the Association for Computational Linguistics, MIT, 26–30th June, 1995, pp. 314–316.

[17] C.I. Guinn, Mechanisms for mixed-initiative human–computer collaborative discourse, in: A. Joshi, M. Palmer (Eds.), Proceedings of the Thirty-fourth Annual Meeting of the Association for Computational Linguistics, Morgan Kaufmann, San Francisco, 1996, pp. 278–285.

[18] P. Callaghan, An evaluation of LOLITA and related natural language processing systems. PhD Thesis. University of Durham, August 1997.

[19] N. Webb, A. De Roeck, U. Kruschwitz, P. Scott, S. Steel, R. Turner, Natural language engineering: slot-filling in the YPA, Proceedings of the Workshop on Natural Language Interfaces, Dialogue and Partner Modeling, at the Fachtagung fur Kunstliche Intelligenz KI'99 at the Fachtagung fur Kunstliche Intelligenz KI'99, Bonn, Germany (1999).

[20] N. Lesh, C. Rich, C. Sidner, Using plan recognition in human–computer collaboration, Proceedings of the Seventh International Conference on User Modeling, Banff, Canada, June 20–24, 1999, pp. 23–32.

[21] R. Mooney, Learning for semantic interpretation: scaling up without dumbing down, in: J. Cussens (Ed.), Proceedings of the First Workshop on Learning Language in Logic, Bled, Slovenia, 1999, pp. 7–15.

[22] T.R. Addis, J.J. Townsend Addis, An introduction to clarity: a schematic functional language for managing the design of complex systems, International Journal of Human–Computer Studies 56 (2002) 331–374.

[23] M. Kantrowitz, Bibliography of Research in Natural Language Generation, Research Working Paper CMU-CS-93-216, Department of Computer Science, CMU, Pittsburgh, PA, 1993.

[24] G. Varile, A. Zampolli (Eds.), Survey of the State of the Art in Human Language Technology (Studies in Natural Language Processing), Cambridge University Press, Cambridge, 1998.

[25] J. Backus, The syntax and semantics of the proposed international language in Zurich, ACM-GAMM Conference, Proceedings of the International Conference on Information Processing UNESCO, June, 1959, pp. 125–132.

[26] J.R. Hobbs, Monotone decreasing quantifiers in a scope-free logical form, in: K. van Deemter, S. Peters (Eds.), Semantic Ambiguity and Underspecification. CSLI Lecture Notes No. 55, Stanford, California, 1995, pp. 55-76.

[27] J. Welsh, J. Elder, Introduction to Pascal, second ed., Prentice Hall, London, 1982.

[28] N. Dale, C. C. Weems, Introduction to Pascal and Structured Design, fourth ed., Jones and Bartlett, 1996.

[29] B.W. Kerningham, D.M. Ritchie, The C Programming Language, second ed., Prentice Hall, Englewood Cliffs, NJ, 1988.

[30] R. Schwartz, T. Phoenix, Learning Perl, third ed., O'Reilly, 2001.

[31] K.D. Voll, T.P. Yeh, V. Dahl, An assumptive logic programming methodology for parsing, International Journal on AI Tools 10 (4) (2001) 573–588.

[32] F. Balena, Programming Microsoft Visual Basic 6.0 (Mps), Microsoft Press, 1999.

[33] B. Stoustrup, The C++ Programming Language, special third ed., Addison-Wesley, Reading, MA, 2000.

[34] D. Flanagan, Java in a Nutshell, fourth ed., O'Reilly, 2002.

[35] A. Goldberg, D. Robson, Smalltalk 80: The Language, Addison-Wesley, Reading, MA, 1989.

[36] G. Gazar, C. Mellish, Natural Language Processing in PROLOG: An Introduction to Computational Linguistics, Addison-Wesley, Reading, MA, 1989.

[37] I. Bratko, Prolog Programming for Artificial Intelligence, third ed., Addison-Wesley, Reading, MA, 2001.

[38] G. Gazar, C. Mellish, Natural Language Processing in Lisp: An Introduction to Computational Linguistics, Addison-Wesley, Reading, MA, 1989.
[39] P. Graham, ANSI Common LISP, first ed., Prentice Hall, Englewood Cliffs, NJ, 1995.
[40] G. Gupta, V. Santos Costa, Complete and efficient methods for supporting side effects and cuts in And–Or parallel Prolog, Proceedings of IEEE International Symposium on Parallel and Distributed Processing, IEEE Computer Society Press, 1992, pp. 288–295.
[41] D.A. Turner, Recursive equations as a programming language, in: J. Darlington, P. Henderson, D.A. Turner (Eds.), Functional programming and its Applications, Cambridge University Press, 1982.
[42] P. Henderson, Lispkit Lisp: purely functional version of LISP, in: P. Henderson (Ed.), Functional Programming, Application and Implementation, Prentice Hall, Englewood Cliffs, NJ, 1980.
[43] S. Thompson, Laws in Miranda, ACM Communications 2 (3) (1986).
[44] R. Bird, Introduction to Functional Programming using Haskell, Prentice Hall, Englewood Cliffs, NJ, 1988.
[45] C. Fellbaum, WordNet: An Electronic Lexical Database, Bradford Books/MIT Press, 1998.
[46] L.M. Iwanska, S.C. Shapiro, Natural Language Processing and Knowledge Representation, MIT Press, Cambridge, MA, 2000.
[47] A. Preece, Evaluating verification and validation methods in knowledge engineering, in: R. Roy (Ed.), Micro-level Knowledge Management, Morgan-Kaufman, San Francisco, 2001, pp. 123–145.
[48] M. Rajman, J. Chappelier, Information extraction out of textual data, Course notes: TIDT/NLP Course (IC-16), AI Lab, Ecole Polytechnique Federale de Lusanne, Switzerland, 2003.
[49] J. Cowie, Y. Wilks, Information extraction, in: R. Dale, H. Moisl, H. Somers (Eds.), Handbook of Natural Language Processing, Marcel Dekker, New York, 2000, pp. 241–260.
[50] J. Hobbs, D. Appelt, M. Tyson, J. Bear, D. Israel, SRI international: description of the FASTUS system, Proceedings of the Fourth Message Understanding Conference (MUC-4), Morgan Kaufmann, Los Altos, CA, 1992, pp. 268–275.
[51] A. Barr, E. Feignbaum, The Handbook of Artificial Intelligence, vol. 1, Pitman Books, London, 1981.
[52] M. Wallace, C. Runciman, The bits between the Lambdas—binary data in a lazy functional language, Proceedings of the International Symposium on Memory Management, Vancouver, Oct (1998).
[53] C. Runciman, N. Rojemo, Heap profiling for space efficiency, Second International School on Advanced Functional Programming, LNCS 1129, Springer, Olympia, WA, 1996, pp. 159–183.
[54] J. Hughes, Why functional programming matters, Computer Journal 32(2) (1989) 1989.
[55] G. Dejong, Prediction and substantiation: a new approach to natural language processing, Cognitive Science 3 (1979) 251–273.
[56] R. Schank, R. Ableson, Scripts Plans Goals and Understanding, Lawrence Erlbaum, Hillside, NJ, 1977, p. 187.
[57] C. Matthews, An Introduction to natural language processing through prolog, Longman (Series: learning about language), 1998.