Constructing Natural Language Interpreters in a Lazy Functional Language

R. FROST[1]* AND J. LAUNCHBURY[2]

[1] *School of Computer Science, University of Windsor, Windsor, Ontario N9B P4, Canada*

[2] *Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland*

In this paper, we describe a method by which language parsers and interpreters may be implemented in a lazy functional programming language. The visual appearance of such interpreters mimics the BNF description of the grammar of the language being interpreted. The method is particularly well suited to the implementation of language interpreters that are based on the principle of 'rule to rule' correspondence (in which each production rule of the grammar has a translation rule associated with it). The main objective of the paper is to demonstrate that the method described provides a useful framework within which both grammars and semantic theories of languages may be investigated. We present the method by example: the simple natural language interpreter we construct is based loosely on principles proposed by Richard Montague.

Received January 1988, revised November 1988

1. INTRODUCTION

Ever since the introduction of definite clause grammars,[4] Prolog has been a natural choice for experimenting with grammars and semantic theories of natural language. In this paper we present a similar scheme for a lazy functional language. The counterparts of definite clause predicates are functions from the input stream to a list of possible parses/interpretations. These functions are combined using higher-order functions to produce composite parsers/interpreters. The higher-order functions are expressed as infix operators, so that the visual appearance of the parser mimics the BNF description of the grammar of the language being interpreted. The result is a clear and modular program which may be easily modified.

We present the method by example. We construct a simple natural-language interpreter that is capable of answering questions about the solar system, its planets and their moons, and the people who discovered the moons. We use a simplistic non-left-recursive grammar that covers a limited subset of English. The semantic theory underlying the interpreter is similar, in some respects, to that proposed by Richard Montague,[3] except that all modal and intensional aspects have been suppressed. In some ways it is more efficient computationally than Montague's semantic theory, being based on set theory rather than on a calculus of characteristic functions of relations.

Following Montague, each word of English is regarded as denoting a semantic object (which may depend on the syntactic category in which the word is used). Each production rule of the grammar has a translation rule associated with it. Using these rules, the meaning of a composite expression is defined in terms of the meanings of its parts.

English sentences can be ambiguous. In Montague's approach, English expressions (both basic and composite) are translated to one or more expressions of an unambiguous language of intensional logic, called IL. The semantics of IL then provide the meaning of the English expression. In our approach we use the unambiguous language of set theory, which we implement in the functional language.

In this paper, we do not intend to argue that the semantics we use to interpret natural language are any better than any other. Rather, we intend to demonstrate that lazy functional languages are suitable for investigating both grammars and semantic theories of language.

1.1 Example session

We translate expressions of English into expressions in the functional language in which the interpreter is written. These are then reduced according to the reduction rules of the functional language, and the results are mapped into English expressions. These are returned to the user as answers to the questions asked. The following is an example of an interactive session with the interpreter.

which planets are orbited by a moon?
earth, mars, jupiter, saturn, uranus, neptune, and pluto.
how many red planets exist?
three.
does every moon orbit a red planet?
no.
mars is a red planet?
true.
which moons orbit mars?
phobos and deimos.
who discovered phobos?
Hall.
did Hall discover deimos?
yes.
which moons were discovered by Kuiper?
miranda and nereid.
which planets are orbited by the moons that were discovered by Kuiper?
uranus and neptune.
does nereid orbit uranus?
no.

* To whom correspondence should be addressed. Each author contributed equally to this work.

nereid orbits neptune?
true.
which moons orbit a solid planet?
luna, phobos, deimos and charon.
every red planet is a gaseous planet?
false.
how many men discovered a moon that orbits jupiter?
six.
which men discovered a moon that orbits jupiter?
Barnard, Galileo, Kowal, Perrine, Nicholson and Melotte.

In the remainder of the paper, we present a (nearly) complete program for a natural-language interpreter written in the notation of Bird and Wadler,[1] but the program requires only minor changes to be run in concrete lazy functional languages such as Miranda or Lazy ML. We have divided the program into four parts. In Section 2 of the paper we present the 'dictionary' of the interpreter; in Section 3 the semantic theory on which the interpreter is based; in Section 4 the method for constructing interpretation functions, together with the particular functions for our example; and in Section 5 we describe how to make the interpreter interactive. We conclude by discussing possible extensions to justify the claim that the method is sufficiently flexible to encourage experimentation.

2. THE DICTIONARY

The dictionary of the interpreter consists of a number of lists of words, paired with their translation. A shortened version is shown in Figs 1, 2 and 3. The structure of the dictionary is dependent on the grammar chosen for the language: there is a separate list for each of the basic syntactic categories of the language to be interpreted. The grammar we use is discussed in Section 4.

Each part of the dictionary is a list of pairs. The first element of each pair is a single word; the second is the translation of the word when used in the given syntactic context. For example, the translation of the word "man" when used as a common noun is the expression commonnoun_man (which we define later).

Words may be used in different syntactic categories. For example, the word "orbit" may be used both as a transitive and as an intransitive verb. Conversely, many words may share the same translation: both "man" and

```
commonnoun =
   [ ("thing",      commonnoun_thing),
     ("things",     commonnoun_thing),
     ("man",        commonnoun_man),
     ("men",        commonnoun_man),
     ("woman",      commonnoun_woman),
     ("women",      commonnoun_woman),
     ("discoverer", meaning_of nounclause "person that discovered something."),
       . . . ]

propernoun =
  [  ("mars",    test_property_wrt 12),
     ("phobos",  test_property_wrt 19),
     ("Barnard", test_property_wrt 55),
     ("Bond",    test_property_wrt 67),
     ("Cassini", test_property_wrt 65),
       . . . ]

relpronoun =
  [  ("that",  relpronoun_that),  ("who",  relpronoun_that),
     ("which", relpronoun_that)  ]

indefinitepronoun =
  [  ("someone",    meaning_of detphrase "a person."),
     ("something",  meaning_of detphrase "a thing."),
     ("somebody",   meaning_of detphrase "a person."),
     ("everyone",   meaning_of detphrase "every person."),
     ("everything", meaning_of detphrase "every thing."),
       . . . ]

adjective =
  [  ("atmospheric", adjective_atmospheric),
     ("red",         adjective_red),
     ("gaseous",     adjective_gaseous),
       . . . ]
```

Note
The symbol . . . indicates more entries

Figure 1. Dictionary – nouns, etc.

```
transverb =

   [ ("discover",   trans_verb rel_discover),

     ("discovered", trans_verb rel_discover),

     ("orbit",      trans_verb rel_orbit),

     ("orbited",    trans_verb rel_orbit),

     ("orbits",     trans_verb rel_orbit)   ]


intransverb =

  [ ("exist",  intransverb_exist),

    ("exists", intransverb_exist),

    ("orbit",  meaning_of verbphrase "orbit something."),

    ("orbits", meaning_of verbphrase "orbit something."),

    ("spin",   intransverb_spin),

    ("spins",  intransverb_spin),

      . . . ]


passtrvb =

  [ ("discovered", passtr_verb rel_discover),

    ("orbited",    passtr_verb rel_orbit)   ]
```

Figure 2. Dictionary – verbs.

"men" are translated to the expression common-noun_man. Alternatively, although we have not given an example, a word could have more than one translation when used in a single syntactic category. One interesting technique we use is to define a word in terms of some phrase. This is achieved by using the function meaning_of. Thus a "discoverer" is a "person that discovers something".

By looking at the dictionary, we can see which words can be used in a query, and also deduce some semantic information. For example, it is easy to see that the system makes no distinction between singular and plural forms of common nouns, nor does it distinguish between the words "a" and "the". In a more realistic system, both of these limitations would be rectified.

Each proper noun is 'associated' with an entity. Entities are abstract objects that have meaning only within the interpreter. In this implementation we represent entities using integers. For example, the proper noun "mars" is associated with the entity represented by the integer 12. Proper nouns correspond to functions that receive a property, and test whether that property is true of the associated entity. The rationale for this approach is discussed in Section 3.

Some words, such as the word "are", are translated to the identity function. This indicates to the user that such words have no effect on the meaning of a composite expression in which they appear other than as a grammatical marker. The fact that, in our example interpreter, the words "are" and "were" are both translated to the identity function may give the (correct) impression that the semantic theory underlying the interpreter does not accommodate time.

3. THE UNDERLYING SEMANTIC THEORY

The semantic theory that we use has some features that were derived from Richard Montague's, but it differs from his in several respects and is much less sophisticated. However, it has the advantage of being simpler to understand, and the interpretation of many English expressions may often be implemented more efficiently. Take, for example, the word "every". In Montague's approach it is translated to a function that takes two characteristic functions of sets as arguments. From these it constructs a new characteristic function, applies this function to all entities in the universe, and conjoins the resulting set of boolean values. In our theory, the word "every" is translated to a set-inclusion test on two sets.

The basic idea, in both approaches, is that English words are translated to expressions of an unambiguous language (according to syntactic category), such that the translation of a composite expression can be obtained from the translations of its parts. This is achieved by

```
linkingverb =

   [ ("is",    id), ("was",   id),

     ("are",   id), ("were",  id) ]


determiner =

   [ ("the",   determiner_a),     ("a",   determiner_a),

     ("some",  determiner_a),     ("no",  determiner_none),

     ("every", determiner_every), ("all", determiner_every),

     ("one",   determiner_one),   ("two", determiner_two),

       . . . ]


termphrasejoin =

   [ ("and", termphrase_and), ("or", termphrase_or) ]


verbphrasejoin =

   [ ("and", verbphrase_and), ("or", verbphrase_or) ]


nounjoin =

   [ ("and", noun_and), ("or", noun_or) ]


preposition =

   [ ("by", id) ]
```

Figure 3. Dictionary – auxiliaries.

associating a simple interpretation rule with each syntax rule (i.e. with each production of the grammar). The semantics of the unambiguous language is then used to obtain, indirectly, the meaning(s) of the English expression.

The difficulty, in building a semantic theory, is in obtaining translations of the basic words such that (1) the grammar is concise, (2) the interpretation rules are simple, and (3) an interpreter based on the theory can be implemented efficiently.

Clearly, the capability of the grammar and semantic theory should depend on the purpose for which the interpreter is to be used. We do not claim that our semantic theory is adequate for anything other than as an example. However, it is plausible enough, as is the unsophisticated grammar on which the interpreter is based.

Our semantic theory is easy to explain. The primitive semantic objects are entities and sets of entities. Then we have functions over these primitive objects. Entities are abstract objects having meaning only within the interpreter. Each entity is, however, 'associated' with individual objects in the real world.

Each common noun denotes the set of entities that may be described by the noun. Adjectives likewise denote those sets of entities that possess the properties the adjectives express, and intransitive verbs are represented by the set of entities for which the verb holds.

Determiners denote boolean valued functions that take two sets as argument. The actual function depends on the determiner. For example, in the phrase "a planet spins" both "planet" and "spins" denote sets. The function corresponding to "a" calculates whether the two sets have a non-empty intersection or not. If the intersection is empty the statement is false, as there is not a planet that spins. Conversely, if the intersection is non-empty there is a planet that spins, so the statement is true. A useful way to view determiners is as 'curried' functions.* That is, if one argument is supplied, the result is a function of the other argument. This approach is useful because it gives meaning to expressions such as "a man". This translates to a function that takes a set-

* Named after Haskell Curry, a logician. If $f: A \times B \to C$ is a function defined on pairs, then the function $curry(f): A \to (B \to C)$ is equivalent except that it receives its arguments one at a time.

valued argument, and returns a boolean to indicate whether the intersection of the set argument with the set of men is non-empty. Thus the meaning of the word "anything" can be defined to be the same as the meaning of the phrase "a thing". We will come across this "currying" or partial application of functions again later.

According to Montague, proper nouns such as "mars" do not denote entities. In fact no expression of English denotes an entity directly. Each proper noun denotes a function that takes some sort of 'property' as an argument and tests whether the property holds for the entity with which the proper noun is 'associated'. This accords with the view that a name such as "mars" does not represent the entity itself, but rather all of the properties that are true of it. It is a view based on analyses of the way in which proper nouns are used.

Similarly, transitive verbs do not denote relations directly, though each is associated to one particular relation. A transitive verb is seen as a function, whose argument is a predicate on sets. When the function is applied to a particular predicate, it returns a set of entities as a result. An entity is in the result set, if the predicate is true of the entity's image under the associated relation.

Relative pronouns and conjunctions such as "and" and "or" are translated to various functions depending on the syntactic category of usage. For example, the word "and" when used to join two verb phrases is translated to set intersection. The variety of definitions is a cost associated with the set-based approach. In Montague's method conjunctions are translated to polymorphic functions whose definitions are independent of the syntactic category.

```
entityset = [1 .. 70]


commonnoun_sun    = [8]

commonnoun_planet = [9 .. 17]

commonnoun_moon   = [18 .. 53]

commonnoun_man    = [54 .. 70]

commonnoun_woman  = [ ]

commonnoun_thing  = [8 .. 70]


adjective_red         = [12, 13, 14, 22]

adjective_blue        = [11, 14, 15, 16]

adjective_ringed      = [13, 14, 15, 16]

adjective_gaseous     = [13, 14, 15, 16]

adjective_solid       = (commonnoun_planet ++ commonnoun_moon)

                                  -- adjective_gaseous

adjective_atmospheric = [10, 11, 12, 22, 42]

adjective_vacuumous   = (commonnoun_planet ++ commonnoun_moon)

                                  -- adjective_atmospheric


intransverb_exist  = entityset

intransverb_spin   = [8 .. 53]


Note

++ is list concatenation

-- is list difference
```

Figure 4. Sets of entities.

```
union as bs       = as ++ (bs -- as)

intersect as bs = as -- (as -- bs)

includes as bs  = (as -- bs) == [ ]


relpronoun_that xs ys = intersect xs ys

termphrase_and p q x  = p x & q x

termphrase_or p q  x  = p x \/ q x

verbphrase_and xs ys  = intersect xs ys

verbphrase_or xs ys   = union xs ys

noun_and xs ys        = intersect xs ys

noun_or xs ys         = union xs ys


determiner_every xs ys = includes xs ys

determiner_a xs ys     = #(intersect xs ys) /= 0

determiner_none xs ys  = #(intersect xs ys) == 0

determiner_one xs ys   = #(intersect xs ys) == 1

determiner_two xs ys   = #(intersect xs ys) == 2


test_property_wrt e ps  = member ps e
```

Note

is list length

& is conjunction

\/ is disjunction

== is the equality test

/= is the inequality test

Figure 5. Implementing sets.

3.1 Implementing the semantics

Figs 4, 5 and 6 give the definitions of the functions used to implement the semantics.

We represent entities by integers. In our example we use the integers 1–70 for the various entities. Sets of entities are implemented using lists of integers. Thus the noun "man" can describe any of the entities from 54 to 70. As it happens, our system doesn't know about any women, so the list of entities that may be described by the word "women" is empty.

As properties are implemented as lists of entities, so proper nouns are translated to functions over lists of entities. In order to test whether a particular property is true of an entity we need only test list membership.

Relations are implemented as lists of pairs. These relations are used directly in the interpretation of transitive verbs. Passive verbs are in some sense the inverse of active verbs. Compare "Hall discovered phobos" with "phobos was discovered by Hall". The order of items related by the passive "was discovered" (using "by" as a placeholder) is the reverse of the order needed for the active "discovered". Passive verbs are implemented exactly as their active counterparts except that they use the inverse of the relation. We obtain this by using the function invert.

The sorts of meanings assigned to words in different syntactic categories vary greatly. Some words are seen as representing sets of entities, others as functions, and so on. These meanings are values, and so each meaning has

```
rel_orbit    = [( 9, 8), (10, 8), (11, 8), (12, 8),
                (13, 8), (14, 8), (15, 8), (16, 8),
                 . . . ]

rel_discover = [(54, 19), (54, 20), (55, 21), (56, 22), (56, 23),
                (56, 24), (56, 25), (57, 26), (57, 34), (58, 27),
                 . . . ]

trans_verb rel p = [x | (x, image_x) <- collect rel; p image_x]
passtr_verb rel  = trans_verb (invert rel)

collect    [ ]   = [ ]
collect ((x,y):t) = (x, y:[e2 | (e1,e2) <- t; e1 = x])
                     :  collect [(e1, e2) | (e1, e2) <- t; e1 /= x]

invert = map swap
        where
          swap (x,y) = (y,x)
```

Note

The functions **trans_verb** and **collect** are defined by *list comprehensions*. These are related to the set comprehensions found in set theory. Replacing the square brackets with curly braces, and the 'drawn from' symbol (<-) with set membership gives a comparable set comprehension.

Figure 6. Implementing relations.

a *type*. The type of the meaning of a word in a particular syntactic category depends only on that category, and not on the word itself. We can, therefore, give a table of the types associated with the various syntactic categories. In the table, ε represents the set of entities, and *Bool* the truth values. Also, if A and B are types then $\{A\}$ is the set of objects of type A, and $A \rightarrow B$ are functions from A to B.

common noun	$:: \{\varepsilon\}$
proper noun	$:: \{\varepsilon\} \rightarrow Bool$
relative pronoun	$:: \{\varepsilon\} \rightarrow (\{\varepsilon\} \rightarrow Bool)$
indefinite pronoun	$:: \{\varepsilon\}$
adjective	$:: \{\varepsilon\}$
transitive verb	$:: (\{\varepsilon\} \rightarrow Bool) \rightarrow \{\varepsilon\}$
intransitive verb	$:: \{\varepsilon\}$
passive transitive verb	$:: (\{\varepsilon\} \rightarrow Bool) \rightarrow \{\varepsilon\}$
determiner	$:: \{\varepsilon\} \rightarrow (\{\varepsilon\} \rightarrow Bool)$

This completes our description of the implementation of individual words. We will study how phrases are handled in the next section.

4. CONSTRUCTING INTERPRETERS

The functions that we shall construct are syntax-directed evaluators. They have a lot in common with parsers, but whereas parsers construct parse trees, we choose to implement evaluation directly. We call these functions 'interpretation functions'.

The method we use has been known to functional programmers for some time. We have tailored the operators for handling natural language – some changes of emphasis would be in order for parsing computer languages. The parsers/interpreters that we construct are equivalent to recursive descent parsers with full backtracking. Ref. 5 contains a detailed discussion of how

```
succeed v inp = [(v,inp)]
fail inp      = []

(p | q) inp = p inp ++ q inp

(p1 --- p2) inp = [((v1,v2), inp2) | (v1,inp1) <- p1 inp;
                                     (v2,inp2) <- p2 inp1]

(p >> fn) inp = [ (fn v, inp') | (v,inp') <- p inp]

item (word,val)    []     = fail []
item (word,val) (wd:wds) = succeed val wds       if  wd == word
                          = fail wds              otherwise

! []     =  fail
!(p:ps)  =  item p  |  !ps


qmark = item ("?","?")
dot   = item (".",".")

meaning_of interp = the_value . (interp --- dot  >> fst) . words

the_value [(v,inp)] = v
```

Note
We will assume that the operators
!   (prefix)
--- (infix, right associative)
>>   (infix, left assoiative)
| (infix, right associative)
are listed in descending binding power

Figure 7. Interpretation primitives.

this is achieved through lazy evaluation. One implication of this is that left-recursive grammars may not be used directly. In this paper we will present the method in a conceptually simple framework without discussing the details of evaluation.

An interpreter is a function: given some input it returns some sort of value as its result. The value is paired with the tail of the input stream so that subsequent interpretation functions can be applied at the point that the first left off. If the grammar is ambiguous more than one value may need to be returned, and there must be a mechanism for returning no value if the input does not match the grammar. We can satisfy these requirements in a uniform way. An interpretation function returns a list of results. The list may be empty (indicating failure), or may contain an arbitrary number of successful interpretations. Each result in the list is a pair consisting of a value and the tail of the input stream.

The most basic interpretation functions are succeed and fail. These play a role analogous to the role 1 and 0 play in natural numbers. From the definitions of the interpretation-function primitives in Fig. 7, we see that the (function-valued) expression (succeed 5) is an interpretation function that will succeed with value 5 whatever the input is. Conversely, fail is an interpretation function that will fail whatever the input is. Even though succeed was defined with two arguments, we can use it with only one. This gives us a function of the remaining argument. This is another occurrence of currying. Uses like this occur many times when constructing interpretation functions.

4.1 Combining interpretation functions

We use four operators to combine interpretation functions, defined in Fig. 7. They are designed to model the form of BNF so that the parser/interpreter explicitly expresses the grammar interpreted. These operators take interpretation functions as arguments, and return an interpretation function as a result. They are therefore higher-order functions. The fact that they are expressed as operators rather than functions is a syntactic feature only. In many functional languages, the user may define prefix and infix operators and give precedence and associativity declarations.

One means of combining items of a BNF grammar is through alternation. Thus, when we want to combine two interpretation functions as alternatives we use an operator |, chosen to mimic the BNF symbol. The result of combining two interpretation functions p and q using | is an interpretation function whose results are *all* the results that either p or q would return. The function fail is a left and right identity for |, that is, fail | p = p = p | fail.

The other major means of combining items of a BNF grammar is through sequencing. In BNF this is written as juxtaposition. Here we use an operator ---. When p1 --- p2 is applied to some input, p1 is applied and returns a list of results. Each of these results contains a tail of the input stream. p2 is applied to each of these tails, and for each one also produces a list of results. The result of p1 --- p2 is a list of pairs. The first component of each pair is itself a pair of values, the first coming from p1 and the second from p2. The second component of each pair is the tail of the input after p2 has finished with it.

The other two operators have no counterpart in BNF. The first allows the values returned by the interpretation functions to be manipulated. We give ≫ an interpretation function on the left, and an arbitrary function on the right. Then in the expression p ≫ fn the function fn is applied to each of the values returned when p is applied to the input. This is particularly useful for combining values obtained using --- when something other than a pair is required. For example, in the grammar we have a clause:

```
sentence = jointermphrase ---
               joinverbphrase ≫ apply2
```

where

```
apply2 (f,x) = f x
```

Suppose that we apply the function sentence to some input value. The interpretation function jointermphrase returns a function as its value, and joinverbphrase a value suitable for that function. So jointermphrase --- joinverbphase returns a pair containing both of these. The operator ≫ applies the function apply2 to each of the result pairs. apply2 takes the function component of its pair and applies it to the value component. The value that sentence returns is the result of the function application.

```
simplenounclause
                =    !commonnoun
                   | adjectives --- !commonnoun                        >> intersct

relnounclause =    simplenounclause --- !relpronoun --- joinvbphrase  >> reorder
                   | simplenounclause

adjectives     =    !adjective --- adjectives                          >> intersct
                   | !adjective

nounclause     =    relnounclause --- !nounjoin --- nounclause         >> reorder
                   | relnounclause

transvbphrase =    !transverb --- jointermphrase                      >> apply2
                   | !linkingverb --- !passtrvb
                          ---   !preposition --- jointermphrase       >> drop3rd

detphrase      =    !indefinitepronoun
                   | !determiner --- nounclause                        >> apply2

termphrase     =    !propernoun
                   | detphrase

verbphrase     =    transvbphrase
                   | !intransverb
                   | !linkingverb --- !determiner --- nounclause       >> drop2nd

jointermphrase=    termphrase --- !termphrasejoin --- jointermphrase  >> reorder
                   | termphrase

joinvbphrase   =    verbphrase --- !verbphrasejoin --- joinvbphrase    >> reorder
                   | verbphrase

sentence       =    jointermphrase --- joinvbphrase                    >> apply2


apply2   (x,y)          = x y
apply3   (x,(y,z))      = x y z
reorder  (x,(y,z))      = y x z
drop2nd  (x,(y,z))      = x z
drop3rd  (w,(x,(y,z))) = w x z
intersct (x,y)          = intersect x y
```

Figure 8. The grammar of the interpreter.

The final operator we use introduces terminals. The operator ! is a prefix, and binds more tightly than the others. Its argument is a dictionary (a list of word/meaning pairs). If the next word in the input is in its dictionary argument, it succeeds and returns the meaning of that word as its value. ! is defined in terms of a function called item that turns a single dictionary item into an interpretation function. If the first word in the input is the word in the pair, item succeeds and returns the value associated with the word. If not, item fails.

Then, given a dictionary, ! turns it into an interpreter for the words contained in the dictionary. If the dictionary is empty, the interpreter will fail when applied to input (return the empty list), otherwise it will try the first word in the dictionary, and then go on and try the rest.

Also in Fig. 7 is the definition of the function meaning_of referred to earlier. The function takes an interpretation function as an argument. The result of this application is another function – one requiring a string input. The string is split up into a list of words, and the interpreter argument is applied to this list. Once the interpreter has finished we look for a fullstop. This forces the interpreter to parse the whole phrase, and not just some initial portion. The function fst then discards the value associated with the full stop. We will only use the function meaning_of on phrases with only one meaning – it would be undesirable to define a word in terms of a meaningless or ambiguous phrase! The function the_value looks for, and returns, the value part of this single result.

4.2 The grammar of the interpreter

The complete grammar that we use is given in Fig. 8. Once again we stress that it is a simple grammar, intended only as an example.

Many of the values returned by the interpretation functions are themselves functions. Typically, the meaning of a clause is obtained from the meanings of its parts by function application. Thus most of the combining functions are just variations of a standard apply function. This exhibits the usefulness of a functional language for this area. If we go to a semantic theory even closer to Montague's, this is even clearer. Montague represented the meaning of many classes of words as lambda terms, and gave rules of combination through application and beta-reduction. To implement this in a functional language is very straightforward. This contrasts with Prolog, where everything must be represented using first-order objects.

To give a flavour of the grammar we will consider an example interpretation. Consider the sentence "phobos orbits mars". In interpreting this:

- sentence looks for a jointermphrase followed by a joinverbphrase.
- "phobos" is a jointermphrase (because it is a termphrase through being in the propernoun dictionary).
- "orbits mars" is a joinverbphrase (via transvbphrase etc.).
- "phobos" is translated to test_property_wrt 19.

```
session inp = introduction
              ++ unlines (map interpret (lines inp))
              ++ conclusion

lines   []     = []
lines (c:cs)   = []:lines cs          if c=='\n'
               = (c:ln):lns           otherwise
                 where
                   (ln:lns) = lines cs

unlines    []    = "\n"
unlines (ln:lns) = ln ++ "\n" ++ unlines lns

introduction = "Hello. I can answer some questions posed in a limited\n\
               \subset of English. My knowledge covers the planets, their\n\
               \moons and discoverers. Please end all questions with a\n\
               \question mark. Use <control-D> to finish.\n\n"
conclusion   = "\n\nGoodbye...\n\n"
```

Figure 9. Session.

- "orbits" is translated to trans_verb rel_orbit.
- "mars" is translated to test_property_wrt 12.
- trans_verb rel_orbit is applied to test_property_wrt 12. The result is the list [19, 20].
- test_property_wrt 19 ("phobos") is applied to [19, 20] (the translation of "orbits mars").
- The result is True.

It is worth noting the direct relationship that the grammar has to its BNF description, and so is both easy to read and to modify. This encourages experimentation.

5. THE INTERACTIVE SESSION

We will model the interactive session as a stream function mapping the input stream to the output stream. This is a very common technique in lazy languages, and works well for many purposes. The value of the input stream becomes available as the user types on the keyboard, and as the output is evaluated it is printed on the screen. The function that maps the input to the output is called session.

The interpretation function is designed to take individual questions and to answer them. Therefore a convenient way to view the input is as a list of lines. We apply the interpret function to each line, and get a line as a result. We need two functions: one to split up the input at the newline characters to give a list of lines; and another to concatenate a list of lines inserting newline characters at the join. We can achieve this by using two fairly standard functions: lines and unlines. The function lines takes a list of characters and divides it at each newline character; unlines does the reverse.

The function session returns the string introduction (which will be printed on the screen), and applies the interpretation function interpret to each line of input. The evaluator pauses at this point as the value of the next line is required. Once it has been entered the evaluator can continue. interpret produces a line in response to each line of input, and unlines turns these lines into a single list of characters. When the user signals 'end of text' by typing control-D, the string conclusion is printed.

How does interpret handle each question? From the definition in Fig. 10 we see that interpret has been written as the composition of other functions. To trace their effect on the input we work from right to left. The line is first split up into a list of words. This list of words is handed to an interpretation function, which looks for a question followed by a question mark. The question mark is discarded by the function fst. What remains is a

```
interpret = disambiguate . map fst . (question --- qmark >> fst) . words


disambiguate   []     = "I do not understand"

disambiguate  [ans]   = ans

else

disambiguate answers = "The question is ambiguous. The possible answers are"

                         ++ concat (map newans answers)

                       where

                         newans a = "\n * " ++ a


words    []       = [[]]

words (c:cs)      = []:words cs          if c==' '

                  = []:[c]:words cs      if c=='.' \/ c=='?'

                  = (c:ln):lns           otherwise

                    where

                      (ln:lns) = words cs


unwords  [x]      = x ++ "."

unwords [x,y]     = x ++ ", and " ++ y ++ "."

else

unwords (x:xs)    = x ++ ", " ++ unwords xs
```

Figure 10. Handling a single question.

```
question =    sentence                                  >> truefalse

         || !doesq --- sentence                         >> apply2

         || !quest1 --- joinvbphrase                    >> apply2

         || !quest2 --- nounclause --- joinvbphrase     >> apply3

         || howmany --- nounclause --- joinvbphrase     >> apply3


howmany  =    !howq --- !manyq                          >> fst


truefalse b = "true."   if b

            = "false."  otherwise
```

Figure 11. Grammar for questions.

```
doesq  = [ ("does", yesno), ("do", yesno), ("did", yesno) ]

quest1 = [ ("who", function_whoq), ("what", function_whatq) ]

quest2 = [ ("which", function_whichq)]

howq   = [ ("how", function_howmanyq) ]

manyq  = [ ("many", id) ]


function_whoq xs         = check "nobody" [name_of e | e <- xs;

                               member (union commonnoun_man commonnoun_woman) e]

function_whatq xs        = check "nothing" [name_of e | e <- xs]

function_whichq xs ys    = check "none" [name_of e | e <- intersect xs ys]

function_howmanyq xs ys = number (# intersect xs ys)


yesno b = "yes."     if b

        = "no."      otherwise


number n = ["none.","one.","two.","three.", . . . ] @ n

name_of e = hd [ name | (name, f) <- propernoun ; f [e] ]


check str wds = str           if wds == [ ]

              = unwords wds   otherwise
```

<u>Note</u>

@ selects the n^{th} element of a list

Figure 12. Question words.

list (possibly empty) of successful interpretations of (that is, answers to) the user's question, paired with the remains of the input line. The map fst converts this into a list of answers only, and disambiguate converts these answers into a single answer.

All that remains is for us to define the interpretation function question (Fig. 11). The user may enter a sentence, to which a true/false response is appropriate – a sentence evaluates to a boolean. Another possibility is to precede a sentence with either "does" or "do". Here a yes/no answer is appropriate, so the "meaning" of a "does"-word in this context is a function that converts booleans to either "yes" or "no". This function is applied to the boolean result of sentence. There are two other possible forms of question also given in the program, but there are others that could have been added. The question form "is...?" is not allowed by the grammar we have given, but it would be a likely candidate for inclusion in an extension.

The remaining functions to handle the other question forms are given. We will not discuss them in detail: the previous discussion should be enough to allow the interested reader to sort out the details.

6. EXPERIMENTATION

Many of the shortcomings of the grammar and semantic theory we have used in the example will be noticed immediately by a linguist. If the grammar were more sophisticated, some of these might not be evident even to an expert reviewing the production and translation rules.

As an example of a shortcoming, consider the following question/answer session:

which moons orbit the planet orbited by miranda?
I do not understand.
which moons orbit the planet that is orbited by miranda?
miranda, ariel, umbriel, titania and oberon.

It is clear, by looking at Fig. 8, that the grammar cannot accommodate relative clauses in which the relative pronoun and the linking verb have been omitted. It may also be seen that the grammar requires more than a simple modification to do this. However, after the designer of the interpreter has modified the grammar, it is a simple matter to edit the program accordingly.

Here is another interesting example.

who discovered a moon that orbits mars or jupiter?
The question is ambiguous. The answers are
* Hall.
* Hall, Barnard, Galileo, Kowal, Perrine, Nicholson and Melotte.

who discovered a moon that orbits mars?
Hall.
who discovered jupiter?
nobody.
who discovered a moon that orbits jupiter?
Hall, Barnard, Galileo, Kowal, Perrine, Nicholson and Melotte.

The fact that the interpreter returns two answers to the first question indicates that the question has been parsed in two ways. The next three questions indicate that the first parse was "who discovered (a moon that orbits mars) or jupiter?" and that the second was "who discovered a moon that orbits (mars or jupiter)?". Both parses are acceptable, but the order in which the answers are presented should perhaps be reversed so that it agrees with the order that most humans would expect. The designer can experiment with the interpreter by changing the order in which alternatives of a production rule are used. For example, the order of the alternatives in the jointermphrase interpretation function could be reversed so that the simpler construct is tried before the more complex. Doing this would solve the problem above.

6.1 Semantics using characteristic functions

The semantics we use in this paper are essentially set-based, using some features from Montague's theory of semantics. This was done to make the presentation simpler, and because nested determiners are handled much more efficiently. However, in doing so we lost some of the elegance inherent in Montague's approach. It is an interesting exercise to take the set-based semantics we use, and to replace occurrences of sets with corresponding characteristic functions. The resulting semantics are much closer in flavour to Montague semantics, and correspondingly more elegant. For example, the two semantic versions of the conjunction 'and' are then unified. These new semantics may then be implemented *directly* within the framework we have already built. The definitions of the single words (as given in the dictionaries) will need to be changed, as will the functions used in the grammar to combine the meanings of the parts of each clause into the meaning of the whole. The fact that the semantics are higher-order presents no difficulty at all. It is worth comparing this situation with that presented in Ref. 2. Here Janssen studies the issues involved in implementing Montague semantics in an imperative language. It is clear that here the translation is cumbersome and complicated, to say the least.

6.2 Extensions

There are other avenues for experimentation. One extension might involve translating words to tuples rather than single functional expressions. The tuples could contain knowledge such as gender, number, sort (e.g. animate object, inanimate object...), etc. More powerful translation rules could then be used to make use of this knowledge to direct the parser. The translation rules could also refer to knowledge stored in sort lattices and knowledge bases. Such an extension would provide a useful framework within which novel approaches to the integration of syntax and semantics could be investigated.

Another direction might be to incorporate modal and intensional constructs, in an attempt to produce more robust and realistic interfaces. This brings us to a final point: the real potential for this method seems to be in providing an *interface* between a database and the real world. A purpose-built relational database is optimised for retrieving information. The role of the interpreter would be to translate complex queries in English to queries in the relational language. The queries would be resolved by the database, and the result passed back to the interpreter to be converted into English again.

7. CONCLUSION

We have given an example of a general method for constructing natural-language interpreters in a lazy functional language. The grammar and semantic theory that we have used have many shortcomings, but were sufficient to give a reasonable example. The basic method itself has similarities with definite clause grammars in Prolog. However, unlike definite clause grammars which are defined as an extension to Prolog, the method can be defined within the functional language.

Given some (non-left-recursive) grammar and a suitable semantic theory to go with it, the construction of an interpreter for the language they define is straightforward. We conclude, therefore, that the method provides a useful framework within which both grammars and semantic theories of language may be investigated.

Acknowledgements

The authors acknowledge the assistance of the funding bodies that provide support for their work, and the university departments to which they belong. Richard Frost is supported by the N.S.E.R.C. of Canada, and is a member of the School of Computer Science at the University of Windsor. John Launchbury is supported by the S.E.R.C. of Great Britain, and is a member of the Department of Computing Science at Glasgow University.

REFERENCES

1. Richard Bird and Philip Wadler, *Introduction to Functional Programming*. Prentice-Hall, Englewood Cliffs, NJ (1988).
2. Theo Janssen, *Foundations and Applications of Montague Grammar*, pp. 335–393. Mathematisch Centrum, Amsterdam (1983).
3. Richard Montague, Formal Philosophy. In *Selected Papers of Richard Montague*, edited R. H. Thomason. Yale University Press, New Haven CT (1974).
4. F. Pereira and D. H. D. Warren, *Definite Clause Grammars compared with Augmented Transition Networks*. Technical Report, Department of Artificial Intelligence, University of Edinburgh (1978).
5. Philip Wadler, How to replace failure by a list of successes. In *Functional Programming Languages and Computer Architectures*, edited J.-P. Jouannaud. Lecture Notes in Computer Science 201, p. 113. Springer-Verlag, Heidelberg (1985).