Using a Functional Language for Parsing and Semantic Processing

Guy Lapalme
Fabrice Lavier
Département d'informatique et de recherche opérationnelle
Université de Montréal
CP 6128, Succ "A"
Montréal Québec Canada
H3C 3J7

July 9, 1997

Abstract

This paper shows how a non-strict functional programming language with polymorphic typing can be used to define grammar rules and semantic evaluation along the lines of Montague. This approach provides a unified formalism needing no preprocessing or postprocessing to the functional language itself: parsing and semantics are declared naturally using function definition and evaluation is done by lambda application. We show that by changing only the model we can, after parsing, compute either the truth value of a sentence or its parse tree.

Keywords: syntax, semantic, parsing, functional programming

1 Introduction

One of the problems to be solved within natural language processing is a smooth integration of semantic and syntactic processing. Logic Programming formalisms like Metamorphosis Grammars or Definite Clause Grammars [20, 10, 23] have achieved a goal of a declarative definition of syntax while integrating an operative definition of semantics by means of "evaluable predicates" or by simulation of semantic processing using meta-interpreters that evaluate data-structures built by the syntactic component.

One of the most fruitful avenue for a declarative definition of semantic processing has been the Montague Grammars [5, 18] approach which relies on lambda calculus (i.e. function definition, manipulation and application). Lambda calculus can be dealt using functional programming languages like Lisp or Scheme. Unfortunately those languages do not really offer a declarative style of programming except by using special purpose interpreters like Augmented Transition Networks [2]. As we are using a pure functional language, it would be possible to translate our approach to a "pure" Lisp subset but a great part of the simplicity and efficiency of our approach rests upon the ease of specification of lazy evaluation and currying inherent in modern typed functional language such as Miranda[25][1] and Haskell[13] where functions are defined by means of recursive equations. Unfortunately, it seems that this approach has been largely ignored by researchers interested in natural language processing.

This paper shows how such functional programming languages can be used to define grammar rules and semantic evaluation along the lines of Montague within a unified formalism needing no preprocessing (like the DCGs) or postprocessing (like a semantic evaluator) to the functional language itself: parsing and semantics are declared naturally using function definition (lambda abstractions) and evaluation is done by lambda application. [27] used this kind of approach to develop the semantics of programming languages.

The next section presents Miranda the functional language we choose for our experiments. Section 3 describes how stream based syntactic analysers can be built in a functional language; it is a very good example of the use of higher order functional programming to obtain a declarative reading of the rules. Section 4 then shows how semantics can be integrated in this setting to obtain a model based semantic evaluator which is illustrated by first giving a model to find the truth value of sentences; then, by changing only the model, we obtain the parse tree of the same sentences. Finally section 5 extends the parser and the semantic evaluators to the case where quantifiers are involved.

2 Presentation of Miranda

Miranda is a functional programming language which features function definition by recursive equations allowing higher order functions (they may accept functions as arguments and may produce functions as results). In our case, following [3] and [26], we represent a parser as a function that accepts a list of tokens and returns either an empty list as indication of failure or a list of results each comprising an interpretation of the tokens accepted (e.g. a parse tree) and the list of tokens remaining in the input. Higher order functions are used to combine parsers to express the fact that two parsers follow each other (sequencing) or that one or the other can be used to match the input (alternation).

Miranda allows the definition of "infinite" structures which are evaluated step by step when a given element of those structures is needed (lazy evaluation). Infinite structures are useful be-

[1] *Miranda* is a registered trademark of Research Software Inc.

cause they enable the incremental building of results even if all the input has not been completed thus modelling easily interactive programs. They also allow an alternative to the backtracking implementation technique for nondeterminism using the "list of successes" approach advocated by [26].

Miranda is a strongly typed language that enables a compile time checking of the program. Miranda allows for polymorphic types, so a single function definition can be used in many contexts while keeping a static checking of the program. This is especially useful to catch "attention errors" that often plague grammars: for example, in Prolog one often uses a predicate with the wrong number arguments or of the wrong "kind" leading to run-time errors which are often difficult to catch.

2.1 Elements of the Miranda programming language

We now give the relevant elements of Miranda that we use for natural language processing. For further details, one should consult [22]. Bird and Wadler[1] provide an excellent introduction to functional programming of this kind. The rest of this section can be skipped by someone already familiar with Miranda or not interested in implementing the basic parsing mechanisms.

2.2 Basic elements

Values are of three primitive types, called `num`, `bool`, and `char`. The type `num` comprises integer and floating point numbers. There are two values of type `bool`, called `True` and `False`. The type `char` comprises the ascii character set; character constants are written in single quotes.

Elements can be grouped into lists by enclosing them in square brackets and separated by commas, for example `[1,4,5,7]`. All the elements of a list must be of the same type. String constants written in double quotes are lists of `char`s, so `"hello"` is only a shorthand way of writing `['h','e','l','l','o']`. Elements of different types can be grouped in tuples by enclosing them in parentheses separated by commas like `(1,"one",True)`.

Function application is noted by juxtaposing the function name with its parameter, for example `f x` indicates applying `f` to `x`. Functions are defined using recursive equations: for example, the following function looks up entries in an association list: given a value `x` and list of tuples `(x',y)`, `search` returns the list of `y`s such that `x=x'`.

```
search x []  =  []
search x ((x',y):rest) =  y: search x rest , x = x'
                       =  search x rest    , otherwise
```

Pattern matching is used to differentiate between cases: the first equation defines the case when the list is empty. The second equation has two alternative right hand sides discriminated by "guards"; a guard is a boolean expression written following a comma or `otherwise` used as a catch all; the alternatives are tried in the order in which they are declared. The colon `(:)` is the infix form of the list constructor; as illustrated in this example, it is used in the left hand side of an equation (in a pattern match) to separate the head from the tail of the list and in the right hand side of the equation to construct a list by adding an element in front of a list. This definition can be paraphrased as follows: search of `x` within an empty list is the empty list; search of `x` given a first pair `(x',y)` is `y` if `x = x'` followed by the search of `x` within the rest; if `x` is different from `x'` then it is the result returned by the search within the rest. This function can be used as follows ($\Rightarrow$ indicates the value returned by the call):

```
search 'd' [('a',1),('b',2),('c',3),('a',4),('b',5),('d',6)]
⇒ [6]

search 'b' [('a',1),('b',2),('c',3),('a',4),('b',5),('d',6)]
⇒ [2,5]
```

Lists can also be defined using "list comprehensions" giving a concise syntax for a rather general class of iterations over lists. A simple example is

```
[capitalize n|n <- "miranda 123"; letter n]
⇒ "MIRANDA"
```

which would be read as "list of all `capitalize n` such that `n` is drawn from the `"miranda 123"` list and it is a letter". We suppose that `capitalize` and `letter` are the functions defined with the "obvious" semantics. List comprehensions often allow clear and compact function definitions. For example, `search` can be defined as

```
search x xys = [y|(x',y)<-xys; x=x']
```

2.3 Function application

Function application is left associative, so when we write `f x y` it is parsed as `(f x) y`, meaning that the result of applying `f` to `x` is a function, which is then applied to `y`. In Miranda every function of two or more arguments is actually a higher order function. This is very useful as it permits partial parameterization (also called currying). For example, `search` is a function accepting a value in parameter and giving back as result another function that searches for that value in an association list. By partially parameterizing `search` we get

```
search_a = search 'a'
```

which is a function looking for pairs having `'a'` as first element. Note that now the list to be searched has to have a character as first element of each tuple.

This possibility is used extensively in our approach because our parsers (which are higher order functions) can thus be created, combined and manipulated while not having to deal explicitely with the input string which is applied only at the end. This takes care of the input and output strings that have to be explicit in Prolog or are generated by the preprocessing of the DCGs.

Now that we have a way to search for an element in a list, we would like to combine those functions to express alternation and sequencing usually found in grammars. To illustrate these combinations, we define another "searcher" function:

```
search_b = search 'b'
```

2.4 Alternation

As a search returns the list of successful elements, alternation is simply the concatenation of the elements of the resulting lists using the `++` operator.

```
search_a_or_b xs = (search_a xs) ++ (search_b xs)
```

That pattern of applying two functions to a list and appending their results could also be "packaged" as

```
alt p q xs = (p xs) ++ (q xs)
```

In Miranda, functions can be used as "infix operators" by preceding the name of the function with `$` so that `p $alt q` is equivalent to `alt p q`. Using currying, we now define along the lines of [6]

```
search_a_or_b = search_a $alt search_b
```

so that

```
search_a_or_b [('a',1),('b',2),('c',3),('a',4),('b',5),('d',6)]
⇒ [1,4,2,5]
```

2.5 Sequencing

Sequentially combining two "searchs" means in fact combining all the results of the first search with the ones of the second. If we take our example of association list search giving back numbers, we could combine the results using addition. Our definition of sequencing is defined using list comprehension as follows

```
sq p q xs = [v1+v2 | v1<-p xs; v2 <-q xs]
```

using infix notation and currying like in the preceding section we now define

```
search_a_and_b = search_a $sq search_b
```

that is now used as

```
search_a_and_b [('a',1),('b',2),('c',3),('a',4),('b',5),('d',6)]
⇒ [3,6,6,9]
```

We thus see that it is possible to combine "searchers" to obtain higher order searchers and this idea will be used later to combine simple parsers into more complex ones.

2.6 Types

Miranda is a strongly typed polymorphic language. Type expressions are used to declare the types of the arguments and the results of functions. We briefly describe the syntax of type expressions:

If `T` is type, then `[T]` is the type of lists whose elements are of type `T`. For example `[[1,2],[3,4],[5]]` is of type `[[num]]`, that is it is a list of lists of numbers. If `T1` to `Tn` are types, then `(T1,...,Tn)` is the type of tuples with objects of these types as components. For example `(True,"hello",36)` is of type `(bool,[char],num)`. If `T1` and `T2` are types, then `T1->T2` is the type of a function with arguments in `T1` and results in `T2`.

Miranda scripts can include type declarations. These are written using `::` to mean *is of type*. Example

```
sq :: num -> num
sq n = n * n
```

Type declarations are not mandatory because the compiler deduces the type of function from its defining equation. It is a good programming practice though because it provides a good documentation for the human reader and allows the compiler to check our definitions and to pinpoint more precisely any discrepancy between what is defined and what was "intended".

Types can be polymorphic, in the sense of [17]. This means that a single definition can be used for arguments of different types provided we are consistent in their use. This is indicated by using the symbols `* ** *** ...` as an alphabet of generic type variables. For example, the identity function, defined in the Miranda library as

```
id x = x
```

has the following type

```
id :: * -> *
```

this means that the identity function accepts as argument a value of any type and it returns a value of the same type. Another example is `search` defined above which has the following type:

```
search::*->[(*,**)]->[**]
```

As `->` is left associative, `search`'s type is thus `*->([(*,**)]->[**])` meaning that `search` accepts a value and returns a function that accepts a tuple having as first element a value of the previous type and as second element a value of possibly another type; this last function returns a list of elements of the second type. In our previous examples, `*` was `char` and `**` was `num`. This is accordance with the partial parameterization of `search_a` which is of type `[(char,**)]->[**]`.

The user may introduce new types using `::=`. For example a type of general n-ary trees would be introduced as follows:

```
tree * ::= Node * [tree *]
```

This introduces two new identifiers - `tree` which is the name of the type, and `Node` which is the constructor for trees. The values stored in the tree can be of any type provided they all are of the same type. `Node` takes two arguments: the value and a list of `tree`s. Using `==`, one can define a synonyms for types; for example, a `string_tree` is a `Node` labelled with a character string.

```
string_tree == tree [char]
```

Here is an example of a tree built using this constructor.

```
Node "P" [Node "N"  [Node "Hank"[]],
          Node "VP" [Node "V" [Node "loves"[]],
                     Node "N" [Node "Mary" []]]]
```

Types with constructors are called algebraic types and can involve many constructors which serve as discriminant between values having component of different types. For example, we can define a binary tree as

```
bin_tree * ::= Nil | Node2 (bin_tree *) (bin_tree *)
```

where the different alternatives are separated by a vertical bar (`|`).

3 Syntactic Analysis

We now show how these tools can be applied for building natural language parsers. Following [3] and [26], a parser is defined as a function that accepts an input pair `(v,xs)` where `v` is a value to be transformed by the parser and an input string `xs`; it returns either an empty list to signal failure or a list of pairs `(v',xs')` where `v'` is the value transformed by the parser and `xs'` the remaining unanalyzed portion of the string. As we deal with natural language processing, the input is a list of list of `char`s (`word`) that is easily built by a lexical analyzer; it is described in appendix. So we define the following type synonyms:

```
word     == [char]
parser * == (*,[word]) -> [(*,[word])]
```

3.1 Building simple parsers

To match terminal symbols (or literals), we define the following

```
literal::(*->word->*)->word->parser *
literal f x (v,[])     = []
literal f x (v,input)  = [(f v x,tl input)]   , x=hd input
                       = []                   , otherwise
```

which is understood as: if `x` occurs at the start of `input` then return a pair comprising the result of the application of the function `f` to the input value `v` and `x`; the second element of the pair is the rest of the string (`hd` is a function returning the first element of a list and `tl` returns the list without the first element). As a literal can only match once at the start of the string then the list of results has only this tuple as element. If the input string is empty or does not start with `x` then return the empty list as a failure signal.

`literal` is very general, but we want our parsers to build a structure of type `string_tree` defined earlier. To achieve that, we create a more specialized version of the parser using a function that returns a leaf of a `string_tree` having `x` as node value.

```
lit::word->parser string_tree
lit    = literal f
         where f w x = Node x []
```

`where` introduces local definitions within a definition. By partially parameterizing `lit`, we get a parser to match a particular word. For example `lit "Sadie"` is a parser that can be applied to list of strings and succeeds with `Node "Sadie" []` if the input starts with "`Sadie`" otherwise it fails by returning the empty list.

```
lit "Sadie" (undef, ["Sadie","snores"])
⇒ [(Node "Sadie" [],["snores"])]

lit "Sadie" (undef, ["Hank","snores"])
⇒ []
```

where `undef` is a dummy value which, in this case, is ignored.

3.2 Combining parsers

3.2.1 Alternation

As we have done for our searchers in the previous section, parsers can be combined for alternation by concatenating the resulting lists.

```
alt p q xs = (p xs) ++ (q xs)
```

Partially parameterizing this combination, we obtain a new parser; for example:

```
alt (lit "Sadie") (lit "Hank")
```

also written as

```
(lit "Sadie") $alt (lit "Hank")
```

is a parser that succeeds if the input starts either with `"Sadie"` or with `"Hank"`.

3.2.2 Sequencing

Sequentially combining parsers is a bit more complex because the output of the first parser becomes the input of the second while combining the results of both. The final output of the sequential combination is the result of the second parser. This behavior can be expressed using list comprehension:

```
sequencing::(*->*->*->*) -> parser * -> parser * -> parser *
sequencing f p q (v,xs) =
          [(f v v1 v2,xs2)|(v1,xs1)<- p (v,xs); (v2,xs2)<-q (v1,xs1)]
```

This is read as: sequentially combining parsers `p` and `q` on input `(v,xs)` is creating of list of pairs composed of the application of `f` to the input value `v` and the values `v1` and `v2` given back by the parsers. The original input is given to `p` and its output `(v1,xs1)` is given to `q` whose output string `xs2` is the output string of the combination. A specialized version of that function to return a `string_tree` is:

```
sq :: parser string_tree -> parser string_tree -> parser string_tree
sq = sequencing f
     where
     f v v1 v2 = Node "@" [v1,v2]
```

`f` combines the two `string_tree`s returned by the parsers by constructing a `string_tree` having `@` as its label to denote function application; it ignores its input argument. As before, we can partially parameterize this function to obtain a new parser; for example:

```
(lit "Sadie") $sq (lit "snores")
```

is a new parser which succeeds if the input starts with the two words `"Sadie"` and `"snores"`. For example:

```
((lit "Sadie") $sq (lit "snores")) (undef,["Sadie","snores"])
⇒ [(Node "@" [Node "Sadie" [],Node "snores" []], [])]
```

3.2.3 Repetition

It is quite often needed to use one parser followed by an arbitrary number of repetition of another one. This is quite useful to transform grammar rules that would otherwise be left recursive. For example, a rule of the form

$$a \quad \rightarrow \quad c \mid a\ b$$

can be transformed into

$$a \quad \rightarrow \quad c\ \{b\}^*$$

but the values are combined on the left using a function f

$$f \ldots (f(f\ c\ b_1)b_2) \ldots b_n$$

Following [3, 21], this transformation can be abstracted into a function using list comprehension as follows:

```
recsequencing::(*->*->*->*) -> parser * -> parser * -> parser *
recsequencing f p q (v,xs)
    = arb_q (p (v,xs))
      where
      arb_q []   = []
      arb_q vxs  = arb_q [(f v v1 v2,xs2)|(v1,xs1)<-vxs;(v2,xs2)<-q(v1,xs1)]
                   ++ vxs
```

which is specialized in the same way as for `sq`.

```
recsq = recsequencing f
        where
        f v v1 v2  = Node "@" [v1,v2]
```

An example of a call is the following

```
(lit "c" $recsq lit "b") (undef,["c","b","b"])
⇒ [(Node "@" [Node "@" [Node "c" [],Node "b" []],Node "b" []],[]),
(Node "@" [Node "c" [],Node "b" []],["b"]),
(Node "c" [],["b","b"])]
```

We get 3 answers because this combination of parsers first returns the longest match and then the other ones having always one match less. In [15], we describe in more detail some of the subtleties involved in the functional approach to parsing using the well known example of arithmetic expressions parsing.

3.2.4 Applying functions

It is sometimes useful to transform the resulting values of a parser using a function. This is simply done with the following which applies a function to each value produced by a parser but it does not change the output string.

```
as::parser *->(*->*)->parser *
as p f xs = [(f v1,xs1)|(v1,xs1)<-p xs]
```

3.3 Writing a full parser

Putting together all these tools, we build and combine parsers in order to check if a list of words conforms to a grammar. We take as first example, the"L0e" language defined in [5, p.23] with the following grammar where terminal symbols are given in quotes and non-terminal ones are in italic.

$$
\begin{array}{lcl}
n & \rightarrow & \text{“Sadie”} \mid \text{“Liz”} \mid \text{“Hank”} \\
vi & \rightarrow & \text{“snores”} \mid \text{“sleeps”} \mid \text{“is\_boring”} \\
vt & \rightarrow & \text{“loves”} \mid \text{“hates”} \mid \text{“is\_taller\_than”} \\
neg & \rightarrow & \text{“it\_is\_not\_the\_case\_that”} \\
conj & \rightarrow & \text{“and”} \mid \text{“or”} \\
vp & \rightarrow & vi \mid vt\ n \\
s & \rightarrow & neg\ s \mid n\ vp \mid s\ conj\ s
\end{array}
$$

The s rule can be rewritten as:

$$s \rightarrow (neg\ s \mid n\ vp)\ \{conj\ s\ \}^*$$

Using the tools defined in the preceding sections, we define our grammar in Miranda:

```
n,vi,vt,ng,conj,vp,s :: parser string_tree

n  = (lit "Sadie") $alt (lit "Liz") $alt (lit "Hank")
vi = (lit "snores") $alt (lit "sleeps") $alt (lit "is_boring")
vt = (lit "loves") $alt (lit "hates") $alt (lit "is_taller_than")
ng =  lit "it_is_not_the_case_that"
conj = (lit "and") $alt (lit "or")
vp = vi $alt (vt $sq n)
s = ((ng $sq s) $alt (n $sq vp)) $recsq (conj $sq s)
```

We see that it is only a matter of transcribing the original grammar and replacing the `|` by `$alt`, the concatenation by `$sq` and repetition by `$recsq`. Now it can be used to parse sentences:

```
s (undef,["Liz","loves","Hank"])
```

giving

```
(Node "@" [Node "Liz" [],Node "@" [Node "loves" [],Node "Hank" []]],[])
```

which represents the following tree:

```
Liz
   loves
   Hank
```

This can be seen if we indent the preceding output as follows:

```
(Node "@" [Node "Liz" [],
           Node "@"    [Node "loves" [],
                        Node "Hank"  []]],
 [])
```

This section has shown how a non-strict functional language can be used for parsing natural language. We used a very simple parsing algorithm which is the classical depth first with backtracking approach as introduced by Burge [3] and Wadler [26]. This technique is now a "standard" practice in many applications of functional programming [24, 14, 21] but to our knowledge we are the first with Frost [7] to apply it in the context of natural language processing instead of the usual artificial (computer) languages. The parsing strategy also corresponds to the analysis strategy used by a standard DCG grammar interpreter in Prolog where unification helps a lot for passing information between components. In Prolog, partially specified structures can be easily constructed but in a functional language we can often achieve much of the same goal by returning a function to be applied later when its argument becomes known. Frost [8] has shown how attribute grammars can be "packaged" using higher-order functions in a lazy functional language.

In our case, parsing is only a small component of our approach, we are more interested in the semantic part along the lines of Montague which is described in the following sections. But as the truth-conditional semantics of a Montague grammar is tightly interconnected with the syntax of the language, it is important to embed parsing and semantic processing in one uniform setting.

4 Semantic Processing

Montague semantics is based on the principle that the meaning of each word can be assigned a function defined in typed lambda calculus. Phrase structure rules are used to combine these basic meanings into new functions which are then evaluated according to a model. As the application of typed lambda abstractions is the fundamental tool of Miranda, it is ideal for implementing Montague semantics.

4.1 Elements of truth-conditional semantics

Dowty[5, p.42] gives the following "essential ingredients" of a truth conditional semantics:

1. "A set of things which can be assigned as semantic values. [...] these are (1) a set of individuals, (2) a set of truth values, (3) various functions contructed out of these by means of set theory."

   In Miranda, these values must all be of the same type; so we define the following algebraic type to encompass all those different cases (`||` indicates the start of a comment that extends until the end of the line):

```
sem_value * ** ::= E * |                    || individual value
                   T ** |                   || truth value
                   Fet (* -> **) |          || functions
                   FeFet (* -> * -> **) |
                   Ftt (** -> **) |
                   FtFtt (** -> ** -> **)
```

   we write it in a very general way so as to allow any type to stand for either an individual value and also for the "truth" value. We define the types of functions involving individuals and truth values. The constructors are chosen here to indicate, using `F` as a prefix "type operator", the type of the corresponding function, for example `Fet` indicates a function from an entity to a truth value and `FeFet` denotes a function from an entity to a function from an

entity to a truth value. This type can be instantiated for specific individuals and choosing boolean values as truth value[2]

```
people ::= A_Sadat | QE_II | H_Kissinger | MMonroe
sem_people == sem_value people bool
```

2. "A specification for each syntactic category of the type of semantic value that is to be assigned to expressions of that category."

As our running example, we take the *L0e* language of Dowty studied in the preceding section on parsing and assign the following types to categories (in fact, all values are of the same type `sem_people` but what differs is the constructor to differentiate the values of the components which are of different types).

category	constructor	type
N	`E`	`people`
Vi	`Fet`	`people -> bool`
Vt	`FeFet`	`people -> people -> bool`
Conj	`FtFtt`	`bool -> bool -> bool`
Neg	`Ftt`	`bool -> bool`
S	`T`	`bool`

3. "A set of semantic rules specifying how the semantic values of any complex expression is determined in terms of the semantic values of its components."

This *Principle of Compositionality* is one of the strength of Montague Semantics and allows the syntax and the semantics to work in tandem. In fact, Montague has defined the way the syntax rules combine the functions for building new ones until we end up with a truth value. For example, the semantics of a verb phrase composed of a transitive verb followed by a noun is given by applying the function associated with the transitive verb `(people -> people -> bool)` to the value associated with the noun `(people)` to give a new function of type `(people -> bool)`; we end up with a new "curried" function of the same type as an intransitive verb, so all verb phrases are of the same type. We define the application rules for combining the semantic values. We use here following predefined Miranda functions: the "dot" operator denoting function composition and `converse f x y` changes the order of the parameters to the function `f` resulting in `f y x`.

```
appff (Fet a)   (E b) = T (a b)
appff (Ftt a)   (T b) = T (a b)

appff (FeFet a) (E b) = Fet (converse a b)    || currying has to
appff (FtFtt a) (T b) = Ftt (converse a b)    || be on the second argument

appff (Ftt a) (Ftt b) = Ftt (b . a)  || function composition

|| symetric cases
```

[2]this last choice seems to be so obvious that it might appear superfluous, but we will later an example of the use of having another kind of "truth value"

```
appff (E b) (Fet a)   = T (a b)
appff (T b) (Ftt a)   = T (a b)
appff (E b) (FeFet a) = Fet (converse a b)
appff (T b) (FtFtt a) = Ftt (converse a b)
```

4. "A specific assignment of a semantic value of the appropriate type to each of the basic expressions."

We define a function that associates a semantic value with each word of the grammar. We give in comments the equation numbers used in pages 25-35 of Dowty [5]. `(&)`, `(\/)` and `(~)` are the standard Miranda functions for the logical *and*, *or* and *not* of boolean values.

```
f0 "Sadie" = E A_Sadat
f0 "Liz"   = E QE_II
f0 "Hank"  = E H_Kissinger
f0 "Mary"  = E MMonroe
||  Vi's
f0 "snores"= Fet snoref                 || (2-13)
             where
             snoref x = x=A_Sadat \/ x=QE_II
f0 "sleeps"= Fet (= A_Sadat)            || (2-14)
f0 "is_boring" = Fet (const True)       || (2-15)
||  Vt's
f0 "loves" = FeFet lovef                || (2-22)
f0 "hates" = FeFet hatef                || (2-26)
             where x $hatef y = ~(x $lovef y)
f0 "is_taller_than"  = FeFet taller_than|| (2-27)
                       where
                       x $taller_than y = height x > height y
                       height A_Sadat = 180
                       height H_Kissinger = 190
                       height QE_II = 170
                       height MMonroe = 175
||  CN's
f0 "man"   = Fet (member [A_Sadat,H_Kissinger])
f0 "woman" = Fet (member [QE_II, MMonroe])
f0 "fish"  = Fet (const False)
||  Conj's
f0 "and"   = FtFtt (&)                  || (2-33)
f0 "or"    = FtFtt (\/)                 || (2-34)
||  Neg
f0 "it_is_not_the_case_that" = Ftt (~)  || (2-31)
||  just in case...
f0 x       = error ("f0>>no semantic for:" ++ x)
||  l'amour toujours l'amour...
||     True if x "loves" y
A_Sadat     $lovef QE_II   = True
```

```
H_Kissinger $lovef MMonroe = True
x           $lovef y       = x=y
```

The above "ingredients" form a *model* defined as

> "a model is an ordered pair $\langle A, F \rangle$ where A is a set of individuals and F is a function which assigns semantic values of the appropriate sort to the basic expression"[5, p. 45]

Model $M0$ of [5, p. 45] is defined as:

```
m =(everybody,f0,show_semantic)

everybody = [E A_Sadat, E QE_II, E H_Kissinger, E MMonroe]

show_semantic :: semantic -> [char]
show_semantic (T x) = show x            || print only truth value
```

where we add, for obvious computational reasons, a function to transform a truth value to a printable one.

4.2 Computing the semantics of a sentence

Montague defines the semantics of a sentence as the truth relative to a model. So our parsers build a function of type

```
appl == model -> sem_people
```

such that when it is applied to the model it computes the truth value. We need to particularize `lit` so that it returns an interpretation function i.e.: a function that given a word and a model, computes a truth relative to a model. Here, we only apply the function to the word.

```
itp :: word->appl
itp x (all,f,show_func) = f x
```

We particularize the parsers as follows: parsing a litteral returns an interpretation function for this word and combining sequentially two parsers means applying these two parsers taking the model into account. `sq` and `recsq` combine two parsers and apply `appff` to their results. This is a form of forward application found in categorial grammars [19].

```
model_parser == parser appl

lit:: [char] -> model_parser
lit = literal f
      where
      f v x = itp x

sq,recsq::model_parser -> model_parser -> model_parser
sq    = sequencing f
recsq = recsequencing f
f v v1 v2 m = appff (v1 m) (v2 m)
```

With these tools, we write the following grammar which follows the same pattern as the one in the previous section. The `s` rule is transformed as before. Here again, thanks to currying, the grammar only deals with the parsing. The interpretation functions of each word are combined incrementally as parsing advances. The truth value is computed by applying the resulting value (`v` in `pp_result`) to the model `m` only when the printing is done.

```
n,vi,vt,ng,conj,vp,s :: model_parser

n = (lit "Sadie") $alt (lit "Liz") $alt (lit "Hank")
vi = (lit "snores") $alt (lit "sleeps") $alt (lit "is_boring")
vt = (lit "loves") $alt (lit "hates") $alt (lit "is_taller_than")
ng = (lit "it_is_not_the_case_that")
conj = (lit "and") $alt (lit "or")
vp = vi $alt (vt $sq n)
s = ((ng $sq s) $alt (n $sq vp)) $recsq (conj $sq s)

|| apply 'sentence parser' on a string and pretty-print results
p = (pp_result m). s' . lex
    where s' x = s (undef, x)

pp_result m x = lay (map (pp_result' m) x)
                where
                pp_result' m (v,x)
                     = show_func (v m) ++ rest
                       where
                       rest = [] , x=[]
                            = ", Unanalyzed:"++ show x , otherwise
                       (all,f,show_func) = m
```

`map f l` applies `f` to all elements of list `l` and returns the list of the resulting values. `lay` transforms a list of strings into a single string where the original strings are now separated by a "newline"; this is only helful for displaying the results. A sentence is parsed and its semantics according to model `m` is found by:

```
p "Sadie loves Liz"
⇒ True

p "Liz loves Hank or Sadie snores"
⇒ True
True, Unanalyzed:["or","Sadie","snores"]
```

The result is a `sem_people` but here it is always a `bool` value because we compute a truth value. The second example illustrates that more than one solution can be obtained and that the truth value can change depending on how far we parse. To evaluate the same sentences but according to another model would only be a matter of redefining another model (`m1` for example) and then printing with `pp_result m1 x`. Thus we achieve a very clear separation between the model and the parser as advocated by Montague.

4.3 Printing the syntax tree

To illustrate the power achieved with this separation between model and parser, we now define a new model that computes a character string such that when it is printed gives the syntax tree of the sentence. The parser is not changed at all.

```
semantic == sem_value string_tree string_tree
m =([],f0,show_semantic)

f0::[char]->semantic
f0 x = E e                , in ["Sadie","Liz","Hank","Mary"]
     = Fet   et           , in ["snores","sleeps","is_boring"]
     = FeFet eet          , in ["loves","hates","is_taller_than"]
     = Ftt   tt           , in ["it_is_not_the_case_that"]
     = FtFtt ttt          , in ["and","or"]
       where
       in ls = member ls x
       nx = Node x []
       e         =  Node "N" [nx]
       et   a    =  Node "@" [a,Node "VI" [nx]]
       eet  a  b =  Node "@" [a,Node "VT" [nx,b]]
       tt   a    =  Node "@" [Node "NG" [nx,a]]
       ttt  a  b =  Node "@" [a, Node "CONJ"[nx],b]

show_semantic::semantic -> [char]
show_semantic (T x) = pp_tree id x
```

`member ls x` is a function that checks if `x` appears in the list `ls`. We can appreciate here the power of the separation between the syntax and the model. The values returned by the parsers are `string_tree` which are then transformed to a more readable form using the pretty print function `pp_tree` defined in appendix.

Now here are a few examples of output using that model with the grammar defined in 4.2.

```
p "Sadie loves Liz"
@ N Sadie
  VT loves
     N Liz

p "Sadie loves Liz or Sadie snores"
@ @ N Sadie
    VT loves
       N Liz
  CONJ or
  @ N Sadie
    VI snores

@ N Sadie
  VT loves
     N Liz
```

```
, Unanalyzed:["or","Sadie","snores"]
```

4.4 Discussion

Montague semantics is often implemented via a special interpreter or processor that comes into play after parsing is completed but here we stay within a unique framework. [11] has shown how to "make computational sense of Montague's Intensional Logic" but in the framework of Lisp: expressions are given procedural interpretations but parsing is not discussed. As Miranda bears a much closer ressemblance with the λ-calculus formalism and this makes our approach more "natural". Frost and Launchbury [7] have also used a lazy functional language to implement a parser and a semantic evaluator inspired by the Montague compositional approach. But their semantics is extensional: in their case, the meaning of a word is a list of integers that serve as indices in a global set of properties. Composition is achieved using set intersection and union. Our approach is more intensional as advocated by Montague: the "meanings" are functions in a typed λ-calculus and new meanings are obtained using function composition.

Another interesting approach is the one taken by Miller and Nadathur [16] in λProlog which defines higher-order definite clauses called λ-terms. This framework enables them to "integrate syntactic and semantic analyses in one computational paradigm"[16, p 247] but the set of functions representable in λProlog is much weaker than the one we use in Miranda because these functions do not represent conditional or recursive definitions. Miller and Nadathur have shown a few examples of use of λProlog for representing the semantics of natural language sentences so they share a common interest with this work. We differ in our interests, they are more concerned with the fundamentals of their higher-order logic in the framework of Prolog while we are focused on the semantics but starting from a functional language.

5 Semantics with individual variables

Dowty[5, p 69-70] introduces *variables* in natural language sentence by means of common nouns (here `man`, `woman` and `fish`) that become bound in the context of quantifiers (here `every`, `some` and `the`). Variables are then refered to by `that` followed by the common noun. For example, the sentence `every man snores` is represented as $\forall v_1 \in \{man\}, v_1 snores$.

We represent our results as follows: the first line is the dictionary of variables in the order of their occurrences in the sentence. The next lines give the quantifiers and their associated variable. Finally, the tree corresponding to the structure of the sentence is given. The common nouns are replaced by the variables bound by the quantifier expressions.

```
variables=["man"]
@ for every v1
  @ v1
    VI snores
```

Before building such trees, we follow the same path taken in the previous section and define a model to evaluate the formulas taking care of the variables. In the next subsection, we will change the model in order to build a syntactic tree.

The parser is similar to the one used in the previous section but it keeps track of the variables between calls to the parsers. To build a semantics taking into account the individual variables, a formula will be interpreted according to a model and, as we need to deal with variables, a variable assignment.

Functions define the assignment of individuals to values. First we use g an initial assignment; g^e_u that "indicates the value assignment exactly like g except that it assigns the individual e to the variable u". To implement this, we define a new function g' that behaves exactly like g but it returns e when its argument is u.

To compute the semantics of a universally quantified variable, we check if for every value assignment `g'` the formula stays *true* i.e. its value is equal to `T True`. The value assignment function is of type `(num -> semantic)` because it indexes the lists of individuals. Note that `(g' e)` is also of that type.

The information returned by a parser consists of the dictionary of variables, the list of quantifiers encountered and the semantic value. An application `appl` is a function from a model and a value assignment that returns a semantic value. A quantifier is a function that transforms an application to another one. These three informations are defined as a tuple `vars_semantic`.

```
assign == num -> semantic
appl == model -> assign -> semantic
dictio == [word]
vars_semantic == (dictio,[appl -> appl],appl)
model_parser == parser vars_semantic
```

We have to modify our interpretation function to take assignments into account. If the word corresponds to a variable then the assignment function is used. `itp x m g` corresponds to $[\![x]\!]^{M,g}$ i.e. "the semantic value of x with respect to M and g"

```
model == ([semantic],[char]->semantic, semantic->[char])

itp::word->appl
itp x (a,f,sf) g = g n  , isvar x
                 = f x , otherwise
                   where
                   isvar [] = False
                   isvar a = digit (last a)
                   n = numval (tl x)
```

We define functions to access the fields of a `vars_semantic`.

```
dic (i,vs,st) = i
vars (i,vs,st) = vs
sem  (i,vs,sm) = sm
```

Once we have finished parsing the sentence, the formula we obtain needs to be evaluated with respect to the quantifiers by combining the quantifiers on the semantic value.

```
eval::(**,[*->*],*)->[*]
eval (d,[],f) = [f]
eval (d,l,f) = [g f|g<-l]
```

We now particularize our basic parsers as follows:

```
lit:: word -> model_parser
lit = literal f
```

```
      where
      f w x = (dic w,vars w,itp x)

sq,recsq::model_parser -> model_parser -> model_parser
sq    = sequencing f
recsq = recsequencing f
f v v1 v2 = (dic v2,vars v2,app2 (sem v1) (sem v2))
app2 f1 f2 m g = appff (f1 m g) (f2 m g)
```

Using these basic parsers, we define parsers for our grammars. As the following parsers do not use variables, they are the same as in the previous section except for `n` and `pn`.

```
n = (pn $alt varia)
pn = (lit "Sadie") $alt (lit "Liz") $alt (lit "Hank") $alt (lit "Mary")
vi = ((lit "snores") $alt (lit "sleeps") $alt (lit "is_boring")) $alt
     (vt $sq n)
vt = (lit "loves") $alt (lit "hates") $alt (lit "is_taller_than")
ng = (lit "it_is_not_the_case_that")
conj = (lit "and") $alt (lit "or")
cn = (lit "man") $alt (lit "woman") $alt (lit "fish")
```

For dealing with quantifiers (*every*, *some* and *the*) and references to the variables (*that*), we use special versions of `lit`. In `quant`, we add a new variable at the end of the dictionary, a new quantifier (using a function `q` which will be described later) at the end of the list of quantifiers and the interpretation function of the variable is given as the value. In `that`, the dictionary is searched for the occurrence of the variable and the name of the variable is given as label; a failure (an empty list) occurs if the variable is not in the dictionary. `shownum` transforms an integer into the string representing it.

```
quant = (lit' "every") $alt (lit' "some") $alt (lit' "the")
        where
        lit' x ((d,l,sem),x:y:input)
           = [((d++[y],l++[q x i (itp y)],itp ('v':shownum i)),input)]
             where i = #d +1
        lit' x y = []

varia = lit' "that"
        where
        lit' x ((d,l,sem),x:y:input)
           = [((d,l,itp ('v':shownum j)),input) | (z,j)<-zip2 d [1..];z=y]
        lit' x y = []
```

A sentence is defined by the rule

$$for \quad \rightarrow \quad neg\ for \mid n\ vi \mid for\ conj\ for$$

which is left recursive. As before, we rewrite it to remove left recursiveness to give

$$for \quad \rightarrow \quad (neg\ for \mid n\ vi)\ \{conj\ for\}^*$$

In this work, we take a very simple minded approach to the generation of quantifier scoping: we transform the grammar and then we have functions that generate exhaustively the scopings. It is reminiscent of the "Cooper Storage"[4] mechanism for generating quantifier scopings. We are aware that this ad-hoc approach is limited and we give some of its drawbacks at the end of this section but we are currently working on integrating the approach of Hobbs and Schieber[12] into this parser. The point we make here is that this functional approach to parsing and semantic processing can be adapted to deal elegantly with quantifiers. Rule corresponding to *n vi* now becomes:

$$n\ vi \mid n\ vt\ quant \mid quant\ vi \mid quant\ vt\ quant$$

```
for = for1 $recsq (conj $sq for)
      where
      for1 = (ng $sq for) $alt
             (n $sq vi) $alt
             ((n $sq vt $sq quant) $as app_l1) $alt
             ((quant $sq vi) $as app_l1) $alt
             ((quant $sq vt $sq quant) $as app_l2)
      app_l1 (d,v,f) = (d,v',f)
                       where
                       v'= [g.(last v)|g<-init v] , init v~=[]
                         = v                      , otherwise

      app_l2 (d,v,f) = (d,v',f)
                       where
                       v'= concat [[g.g2.g1,g.g1.g2]|g<-init (init v)] , #v>2
                         = [g2.g1,g1.g2]           , otherwise
                       g2=last v
                       g1=last (init v)
```

`app_l1` and `app_l2` build all possibilities of combining two quantifiers in a sentence, `app_l1` being the default case, where quantifiers are read left to right, and `app_l2` accounting for semantic ambiguities, which arise only when there are two quantifiers present in the same phrase (as in "every man loves some woman").

The quantifier corresponding to "for some v2 in woman" is now `q "some" 2 (itp "woman")`, of type `appl->appl`. This function, applied to a formula (a function of type `appl` representing the sentence but with its variables not yet assigned), returns an `appl` with its second variable assigned.

```
q quant var prop formula
    = formula'
      where
      formula' (a,f,sf) g
          = T (and s)          , quant = "every"
          = T (or s)           , quant = "some"
          = T (#[t|t<-s;t]=1) , quant = "the"
            where
            s=[y |E x<-a; p x; T y<-[formula (a,f,sf) (g' (E x))]]
            g' z n = z    , n=var
                   = g n , otherwise
```

```
Fet p = prop (a,f,sf) g
```

s is the list of evaluations of the formula according to the model (a, f) and the modified assignment g_{var}^{Vx} for all individuals x that have the property p (i.e. that of being a man, a woman or a fish). The top-level parser function is the following

```
p = (pp_result m g_init). for' . lex
    where
    for' x = for (([],[],undef) , x)

g_init x = error ("v"++shownum x++" undefined")

pp_result m g x = lay (map (pp_result' m g) x)
                  where
                  pp_result' m g (v,x)
                     = show_answer m g v ++ rest
                       where
                       rest = [] , x=[]
                       rest = "Unanalyzed:"++ show x ++ "\n", otherwise

show_answer :: model -> assign -> vars_semantic -> [char]
show_answer m g x
   = concat ["variables=" ++ show(dic x) ++"\n"
             ++(show_func (y m g)) | y<-(eval x)]
     where
     (a,f,show_func)=m
```

Examples of parses are:

```
p "Hank loves some man and that man loves some woman and that woman loves Liz"
variables=["man","woman"]
False
variables=["man","woman"]
False
variables=["man","woman"]
True
Unanalyzed:["and","that","woman","loves","Liz"]
variables=["man"]
True
Unanalyzed:["and","that","man","loves","some","woman","and","that","woman","loves","Liz"]

p "every man loves some woman"
variables=["man","woman"]
False
variables=["man","woman"]
True
```

In this approach, by evaluating only at the end, we implicitly decided that the scope of all quantifiers is the whole sentence. This generally does not raise any problems, except in the case of vacuous

formulas. As the phrase "every fish loves Liz" is vacuously true, because there is no fish in the model, evaluation stops there because there is no assignment possible to the variable associated with "fish", and the rest of the sentence is ignored.

```
p "every fish loves Liz and Hank snores"
variables=["fish"]
True
variables=["fish"]
True
Unanalyzed:["and","Hank","snores"]
```

even though

```
p "Hank snores"
variables=[]
False
```

This approach also fails to take into account the prefered reading in some cases like the following where negation should outscope the second existential and thus should evaluate to `False`.

```
p "some man snores and it_is_not_the_case_that some man snores"
variables=["man","man"]
True
variables=["man"]
True
Unanalyzed:["and","it_is_not_the_case_that","some","man","snores"]
```

5.1 Syntax tree with variables

Like in the previous section, computing the syntax tree is only a matter of changing the model. The truth value is a `string_tree`; for each word, the interpretation function `f1` computes such a tree with the appropriate label. The `semantic` and the new model `m` are defined as follows:

```
semantic == sem_value string_tree string_tree
m =([],f1,show_semantic)

show_semantic::semantic -> [char]
show_semantic (T x) = pp_tree id x
show_semantic x     = error ("show_semantic>>"++show_sem x)

f1::[char]->semantic
f1 x = E e                , in ["Sadie","Liz","Hank","Mary"]
     = Fet   et           , in ["snores","sleeps","is_boring"]
     = FeFet eet          , in ["loves","hates","is_taller_than"]
     = Ftt   tt           , in ["it_is_not_the_case_that"]
     = FtFtt ttt          , in ["and","or"]
     = E e'               , in ["man","woman","fish"]
     = E e''              , in ["every","some","the"]
       where
       in ls = member ls x
```

```
nx = Node x []
e          =  Node "N" [nx]
et   a     =  Node "@" [a,Node "VI" [nx]]
eet  a  b  =  Node "@" [a,Node "VT" [nx,b]]
tt   a     =  Node "@" [Node "NG" [nx,a]]
ttt  a  b  =  Node "@" [b, Node "CONJ"[nx],a]
e'         =  Node "CN" [nx]
e''        =  Node "QUANT" [nx]
```

For a quantifier, we associate a function that adds a new level of `Node` to the tree corresponding to a formula. This new `Node` is labelled with a string comprising the word `for`, the quantifier itself and the number of the current variable.

```
q quant var prop formula
   = formula'
     where
     formula' m g
        = T (Node "@" [Node ("for "++ quant ++ " v"++shownum var)[],
                       f])
          where
          T f = formula m g

g_init x = E (Node ('v':shownum x) [])
```

Example of parses are:

```
p "Sadie loves the man"
variables=["man"]
@ for the v1
  @ N Sadie
    VT loves
       v1

p "Hank loves some man and that man loves some woman and that woman loves Liz"
variables=["man","woman"]
@ for some v1
  @ for some v2
    @ @ @ N Hank
          VT loves
             v1
        CONJ and
        @ v1
          VT loves
             v2
      CONJ and
      @ v2
        VT loves
           N Liz
```

```
 .... and 3 other possible parses

p "every man loves some woman"
variables=["man","woman"]
@ for some v2
  @ for every v1
    @ v1
      VT loves
         v2
variables=["man","woman"]
@ for every v1
  @ for some v2
    @ v1
      VT loves
         v2
```

6 Conclusion

This paper has shown how a functional language can be used for an extensional model theoretic treatment of an English fragment. This is not the full Montague semantics but we are currently working on the implementation of tense and modal operators to come finally to the intensional logic. Any implementation has to deal with finite domains and the use of a functional language certainly does not remove that "restriction" but we are convinced that the basic framework we described in this paper provides a natural and useful experimental "workbench" for Montague semantics.

The functional has some advantages compared to logic grammars[20, 10, 9]: we have a unified functional framework for dealing with lexical, syntactic and semantic processing. But it has some drawbacks for instance in a logic grammar: for instance, unification in a logic grammar is very useful for maintaining undirectional constraints between as yet unspecified values and this behavior can become quite complex to simulate in a functional framework where the only tool we have to deal with this problem is to postpone the evaluation until it is strictly necessary.

We have only shown here how to evaluate a formula according to a model and this is accordance with the original idea of Montague semantics but one of the important goal of natural language processing is the possibility of deducing some new facts. Frost and Launchbury [7] have shown how a functional lazy language can be useful in this aspect and we intend to pursue in that direction.

The parsers shown here are a stream-based implementation of top-down backtrack parsing following the ideas of Burge[3] and Wadler[26] but are not the most efficient implementation of context-free parsing. We are quite sure that it would be simple to implement a non-backtrack parser in a functional language and in this way get an order of magnitude in speed-up.

The grammars given here are still "toy" examples and do not address all the more interesting problems of natural language processing such as scoping and anaphora. Our grammar do not use any features or movement rules and does not have a very sophisticated semantics; we would like to be able to generate logical forms or take indexicality into account. But we are only beginning the exploration of the functional approach to parsing and semantic processing which we believe to be very promising.

7 Acknowledgements

We would like to express our appreciation to Richard Frost and François Lepage for fruitful discussions on this topic of functional languages for semantic processing. We also thank Laurent Trilling who gave us the insight that functional values could be used in many more ways for parsing than we had first anticipated.

References

[1] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[2] L. Bolc. *The Design of Interpreters, Compilers and Editors for Augmented Transition Networks.* Springer-Verlag, 1983.

[3] W.H. Burge. *Recursive Programming Techniques.* Addison-Wesley, 1978.

[4] Robin Cooper. *Quantification and Semantic Theory.* Reidel, Dordrecht, 1983.

[5] D.R. Dowty, R. E. Wall, and S. Peters. *Introduction to Montague Semantics.* Studies in Linguistics and Philosophy. D. Reidel, 1985.

[6] J. Fairbairn. Making form follow function: an exercise in functional programming style. *Software-Practice and Experience*, 17(6):379–386, 1987.

[7] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional language. *Computer Journal*, 32(2):108–121, 1989.

[8] R. A. Frost. Use of algebraic identities in the calculation of programs constructed as executable specifications of attribute grammars. School of Computer Science, University of Windsor, october 1989.

[9] A. Gal, G. Lapalme, and P. St-Dizier. *Prolog pour l'analyse automatique de la langue naturelle.* Eyrolles, 1988.

[10] Gerald Gazdar and Chris Mellish. *Natural Language Processing in PROLOG.* Addison Wesley, 1989.

[11] J. R. Hobbs and S.J. Rosenshein. Making computational sense of Montague's intensional logic. *Artificial Intelligence*, 9:287–306, 1978.

[12] Jerry R. Hobbs and Stuart M. Shieber. An algorithm for generating quantifier scopings. *Computational Linguistics*, 13(1-2):47–63, January-June 1987.

[13] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990.

[14] Graham Hutton. Parsing using combinators. In Kei Davis and John Hughes, editors, *Functional Programming, Glasgow 1989*, Workshops in Computing, pages 353–370. Springer-Verlag, Aug 1989 1990.

[15] Guy Lapalme and Fabrice Lavier. Using a functional language for parsing and semantic processing. Publication 715a, Département d'informatique et de recherche opérationnelle, Université de Montréal, 1990.

[16] Dale A. Miller and Gopalan Nadathur. Some uses of higher-order logic in computational linguistics. In *Proceedings of the 24th Annual Meeting of the ACL*, pages 247–256, June 1986.

[17] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[18] R. Montague. The proper treatment of quantification in ordinary english. In *Approaches to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics.* D. Reidel, 1973.

[19] R. T. Oerhle, E. Bach, and D. Wheeler. *Categorial Grammars and Natural Language Structures.* Kluwer Academic Publishers, 1988.

[20] F.C.N. Pereira and S.M. Shieber. *Prolog and Natural Language Analysis.* CSLI Lecture Notes #10. Stanford University, 1987.

[21] Chris Reade. *Elements of Functional Programming.* International Computer Science Series. Addison Wesley, 1989.

[22] Research Software Limited. *Miranda System Manual.* 1987.

[23] Patrick Saint-Dizier and Stan Szpakowicz, editors. *Logic and Logic Grammars for Language Processing.* Series in Artificial Intelligence. Ellis Horwood, 1990.

[24] Simon Thompson. Interactive functional programs. In David A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, pages 249–285. Addison-Wesley, 1990.

[25] D.A. Turner. Miranda - a non strict functional language with polymorphic types. In P. Jouannaud, editor, *Conference on Functional Programming and Computer Architecture, Lecture Notes in Computer Science #201*, pages 1–16, 1985.

[26] P. Wadler. How to replace failure by a list of successes. In P. Jouannaud, editor, *Conference on Functional Programming and Computer Architecture, Lecture Notes in Computer Science #201*, pages 114–127, 1985.

[27] D.A. Watt. Executable semantic descriptions. *Software-Practice and Experience*, 16(1):13–43, 1986.

8 Appendix

8.1 Lexical analyzer

A lexical analyzer is easily specified in a functional language as can be seen with the following program which decomposes a string in a list of words consisting of letters and underlines. Here characters that are not letters or underlines are ignored but they signal the end of a word.

```
lex :: [char] -> [word]
lex []      = []
lex (x:xs)  = (x:word):lex rest    , inword x
            = lex xs               , otherwise
              where
              word = takewhile inword xs
              rest = dropwhile inword xs
              inword x = letter x \/ x='_'
```

8.2 Pretty printing a tree

The following code pretty prints the contents of a `tree` where `show_star` is a function to transform an element of a `Node` into a string. It uses the auxiliary definition `pp_tree'` which does a recursive descent of the tree keeping track of the current indentation and adding the appropriate number of blanks (using the `spaces` predefined function) in front of each line.

```
pp_tree::(*->[char])->tree *->[char]
pp_tree show_star = pp_tree' show_star 0

pp_tree' show_star indent (Node value leafs)
                = label ++ "\n"                       , leafs=[]
                = label ++ " " ++ pp_tree' show_star nindent (hd leafs) ++
                  concat
                    (map ((spaces nindent ++).(pp_tree' show_star nindent))
                                  (tl leafs))               , otherwise
                  where
                  label   = show_star value
                  nindent = indent+(#label + 1)
```