Realization of Natural Language Interfaces Using Lazy Functional Programming

Richard A. Frost

University of Windsor

The construction of natural language interfaces to computers continues to be a major challenge. The need for such interfaces is growing now that speech recognition technology is becoming more readily available, and people cannot speak those computer-oriented formal languages that are frequently used to interact with computer applications. Much of the research related to the design and implementation of natural language interfaces has involved the use of high-level declarative programming languages. This is to be expected as the task is extremely difficult, involving syntactic and semantic analysis of potentially ambiguous input. The use of LISP and Prolog in this area is well documented. However, research involving the relatively new lazy functional programming paradigm is less well known. This paper provides a comprehensive survey of that research.

Categories and Subject Descriptors: A.1 [**Introductory and Survey**]; D.l.l [**Programming Techniques**]: Applicative (Functional) Programming; J.5 [**Arts and Humanities**]: *Linguistics*; I.2.1 [**Artificial Intelligence**]: Applications and Expert systems—*Natural language interfaces*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Natural language*; I.2.7 [**Artificial Intelligence**]: Natural Language Processing—*Language models*; *Language parsing and understanding*; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems—*Grammar types*; *Parsing*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; *Syntax*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Denotational semantics*; *Partial evaluation*; H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms: Languages, Human factors

Additional Key Words and Phrases: Natural-language interfaces, lazy functional programming, higher-order functions, computational linguistics, Montague grammar

ACM Reference Format:

Frost, R. A. 2006. Realization of natural language interfaces using lazy functional programming. *ACM Comput. Surv. 38*, 4, Article 11 (Dec. 2006), 54 pages. DOI = 10.1145/1177352.1177353 http://doi.acm.org/10.1145/1177352.1177353

The Natural Science and Engineering Research Council of Canada (NSERC) provided financial support for this work.

Author's address: University of Windsor; email: rfrost@cogeco.ca.

1. INTRODUCTION

The range of expressions that can be analyzed using linguistic theories of natural language is far larger than the range of expressions that can be processed by currently available computer-based natural language interfaces (NLIs). The challenge of building computer programs to implement linguistic theories remains and will continue as new theories are developed to accommodate even more aspects of natural language.

Research on NLIs has a long history. Much of the work has involved the use of high-level declarative languages such as LISP and Prolog. That work is well documented in research publications, textbooks, and university course material. The more recent use of lazy functional programming (LFP) in this problem area is less well known and is the subject of this survey.

A functional program consists of a set of function definitions. Execution involves applying functions to their arguments. In pure functional programming, function composition and function application are the only forms of control construct. There are no `for` loops, `while` loops, or `gotos`, and iteration can only be achieved through recursive function calls. There is no updateable state and no notion of imperative command such as changing the value of a variable. The advocates of pure functional programming argue that these constraints lead to highly-modular programs that are easy to analyze, transform, and reuse.

One form of pure functional programming is called lazy functional programming (LFP). Informally, the evaluation of arguments to functions in a lazy language is delayed until those values are required. This is equivalent to normal-order evaluation in the lambda calculus. LFP languages are usually polymorphically strongly typed and come with automatic static type checkers.

Several papers discussing the features, implementation, and application of the LFP paradigm appeared in a special issue of The Computer Journal edited by Wadler [1989]. Since then, over 45 researchers have investigated and published results on the use of lazy functional programming in natural language interface design and implementation. Their work appears to have been prompted by recognition of the similarities between some theories that have been proposed for natural language processing and the theories on which LFP is based and also by recognition of the potential that LFP has in this difficult problem area.

Some of the researchers are affiliated with groups at the Department of Computing Science at Chalmers University in Gothenburg, the CWI Research Institute in the Netherlands, the Department of Computer Science at the University of Durham in the U.K., the Department of Computer Engineering at the Middle East Technical University in Ankara, the School of Computer Science at the University of Windsor in Canada, and the Department of Computer Science at Yale. Collectively, these researchers have published over 60 papers in journals and refereed conference proceedings, which are directly related to the use of LFP in NLP. Other researchers have published an equal number of papers on developments in LFP which have important consequences for NLIs (such as the implementation of generalized LR parsers).

This survey provides a comprehensive review of research in this area. It is intended to be read by Computer Scientists and Computational Linguists. In order to make the descriptions accessible to both groups, the survey begins with background descriptions and references; Section 2 contains a discussion of the difficulties in building natural language interfaces; Section 3 contains brief descriptions of those theories of natural language that have been developed by linguists and which have been referred to in the use of LFP in NLIs; and Section 4 contains a brief introduction to the notation and features of LFP languages. Parts of this introductory material can be skipped by those who are already familiar with them. The latter part of the survey contains relatively

detailed descriptions of research on the use of lazy functional programming in the design and implementation of natural language interfaces.

Throughout the survey, we use longer forms of definitions of logical expressions and program code than would be used by experts in the area. We have chosen to do this in order to make the survey more widely accessible. In particular, we have used only the basic syntax of the Haskell programming language in our examples. Although this may be a little frustrating for some, it does allow the definitions to be easily read by others with less experience of this particular lazy functional programming.

2. THE CHALLENGE OF BUILDING NATURAL LANGUAGE INTERFACES (NLIS)

Natural language interfaces are found in many applications for example, database query and information retrieval systems, language translation systems, Web search engines, dictation systems, and command and control systems. We begin by discussing the difficulty of building natural language database query processors.

Consider evaluating the query "How many students with a g.p.a. greater than 12 are enrolled in course 60–454?" One approach is to translate the query to an expression of a formal query language such as SQL and subsequently execute the query against a database. This approach has limited application owing to an inability to handle negation, modality, and intensionality in queries such as "Does John believe that the Prime Ministers of England and Australia have never met?".

A potentially more-powerful approach is to analyze the user input and compose a response from the meanings of its component substructures (rather than translating it into another language). In applications where a large number of expressions are possible, the rules that are used to compute the response are often applied according to the syntactic structure of the input. Such syntax-directed evaluation is regularly used in the processing of programming and other formal languages. However, application of this technique to natural language is not straightforward. Consider the simple queries "Does Phobos spin?" and "Does every moon spin?". One approach is for proper nouns, such as "Phobos" to denote entities, and intransitive verbs and common nouns such as "spin" and "moon" to denote sets of entities. The two queries above could then be evaluated using rules such as the following, where $\|x\|$ represents the denotation (meaning) of x:

```
query ::= Does proper_noun intransitive_verb
answer = True, if ‖proper_noun‖ ∈ ‖intransitive_verb‖
       = False, otherwise

query::= Does every common_noun intransitive_verb
answer = True, if ‖common_noun‖ ⊆ ‖intransitive_verb‖
       = False, otherwise
```

Now consider extending these rules to accommodate queries such as "Does Phobos and every planet spin?" Do we need to define a new rule for this and every other new type of query? If so, we will need hundreds of rules for even a relatively small NL query language. The solution is to find a small grammar that covers the query language, and a matching semantic theory which assigns a single semantic rule to each of the syntax rules in the grammar. This is not an easy task even for NL query interfaces to first-order relational databases.

The difficulty of building adequate sets of syntactic and associated semantic rules is compounded by 1) ambiguity in phrases such as "Is every planet orbited by a moon?" and "Is Mary a beautiful dancer?", 2) intensionality in queries such as "Did the prime ministers of England and Australia ever meet?", 3) modality in queries such as "Does

John believe that Phobos orbits Venus?", and 4) negation in queries such as "Is Mars orbited by no moon".

Many natural language database query processors have been developed over the last forty years. A good survey of the work, up to 1995, is given in Androutsopoulos et al. [1995]. Since 1995, there has been an annual conference on applications of natural language interfaces to databases, for instance, Meziane and Metais [2004].

Despite the difficulty of constructing comprehensive natural language database-query processors as discussed previous, they are arguably, the simplest type of NLI due to the fact that the data which is to be used in interpreting queries is circumscribed by the first-order relational format and content of the database. Information retrieval systems, on the other hand, have equally complex natural language input that has to be interpreted with respect to knowledge represented in a variety of formats, including ambiguous poorly structured text. Natural language translation is also significantly more difficult as it involves conversion between two languages where the input language may consist of expressions that do not contain all of the information necessary to produce an expression in the target language. For example, when translating from English to German, additional gender information may have to be deduced or added by human intervention.

Review of the literature shows that existing NLIs can only accommodate a small subset of the expressions that can be explained by existing linguistic theories. One of the reasons for this is that efficient implementation of those theories is a nontrivial task, and much work remains to be done. The objective of this survey is to show how lazy functional programming is contributing to this task.

3. RELEVANT THEORIES OF NATURAL LANGUAGE

Three major goals of linguistic theories are: 1) to explain why some sequences of words constitute expressions (sentences or phrases) of a natural language, whereas other sequences do not, 2) to make explicit the syntactic (grammatical) structure of expressions, and 3) to explain how semantic values (meanings) can be ascribed to expressions.

Good linguistic theories are compositional in the sense that they can accommodate large subsets of natural language with as few and as simple rules as possible. The rules are chosen to be highly orthogonal in the sense that they are applicable in many contexts. A principle of correspondence between syntax and semantics is often adopted in which expressions of the same syntactic category denote semantic values of the same type. Also, a principle of rule-to-rule correspondence is often used in which there is a homomorphism between the rules which show how composite expressions are formed from their components, and the rules which show how the meanings of those composite expressions are computed from the meanings of their components. (A homomorphism is a many-to-one structure-preserving function). Such compositionality is also the goal of the denotational semantics approach to the specification of programming languages [Stoy 1977].

Numerous linguistic theories have been proposed, and the literature on the subject is immense. In this section, we review only those theories that have been referred to extensively in research on the use of LFP in NLIs: Finite state automata, context-free and context-sensitive grammar, Categorial Grammar, Montague Grammar, Combinatory Categorial Grammar, type-logical grammar, and type-theoretic grammar. Other theories that have been developed are summarized later in sections which describe their implementations in LFP. Note that we capitalize first letters of some grammar names and not others according to common usage.

The descriptions are necessarily brief and are intended only to provide readers with sufficient background for the discussions in sections 5, 6, 7, and 8. References to more

complete accounts are given in the text. In addition, the book *Type-Logical Semantics* [Carpenter 1998] is recommended for comprehensive coverage of both the linguistic and mathematical theories referred to in this survey.

3.1. Finite State Automata, Context-Free and Context-Sensitive Grammar

We begin with a short review of topics that are familiar to computer scientists: finite-state automata (FSA) and context-free grammar (CFG). Although these systems are not of great interest to linguists (for various reasons, some of which are mentioned later), they serve as a reference in describing other approaches that are discussed later in this section. In addition, CFGs are used extensively in the construction of NLIs where their limitations in explaining a wide range of natural language features are counterbalanced by the ease with which CFG processors can be implemented.

A finite automaton consists of an input alphabet (set of symbols), a start state, a set of accept states, and a transition function which defines the rules for moving from one state to another when an input symbol is read. A string of input symbols is accepted by an FSA if, when reading that string, the FSA moves from the start state to one of the accept states. The language of an FSA is the set of all strings that it accepts. FSAs have various uses in computational linguistics [Roche and Schaber 1997] including analysis of morphology (i.e., the structure of words), information extraction from text, and part-of-speech tagging. However, they are not well suited for defining the syntax and semantics of expressions of natural language. One reason is that, although FSAs can determine (recognize) if a string belongs to a language, they do not facilitate generation of the syntactic structure of that string. Other reasons are discussed in, for example, Copestake [2005].

Context-free grammar was first used to define the syntax of natural languages by Chomsky [1957]. It was also discovered independently and applied to the definition of programming languages by Backus [1959] and Naur et al. [1960]. A context-free grammar consists of four components:

```
T, a set of terminals - words of the language.
N, a set of non-terminals - names for syntactic categories.
P, a set of production rules of the form n::= y, where n is a single
    non-terminal and y is a sequence of zero or more symbols from T ∪ N.
S, a start symbol - a member of N.
```

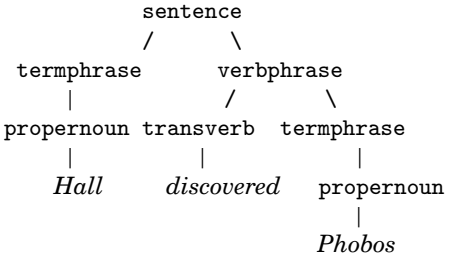
Sequences of terminals can be derived from any nonterminal by repeatedly applying the production rules as left-to-right rewrite rules until the derived sequence contains no nonterminals. Any sequence of terminals that can be derived from the start symbol is called a sentence. The set of sentences so derived is the language generated by the grammar. The following is an example of a CFG for a tiny fragment of English, where `x ::= y | z` is shorthand for the two rules `x ::= y` and `x ::= z`. In this and subsequent examples, we use italics for words in the language being defined.

```
sentence         ::= termphrase verbphrase
termphrase       ::= propernoun | determiner noun
propernoun       ::= Mars | Phobos | Deimos | Hall | Kuiper
determiner       ::= every | a
noun             ::= moon | planet | person
verbphrase       ::= transverb termphrase | intransitiveverb
transverb        ::= discovered | orbits
intransitiveverb ::= spin | exist
```

The following is an example derivation, proving that "Hall discovered Phobos" is a sentence in the language defined by the previous grammar:

```
sentence  => termphrase verbphrase      => propernoun verbphrase
          => Hall verbphrase            => Hall transverb termphrase
          => Hall discovered termphrase  => Hall discovered propernoun
          => Hall discovered Phobos
```

In addition to their generative capability, CFGs can be used to recognize (determine) if sequences of terminals are sentences of a language and also to parse (syntactically analyze) sequences of terminals and assign a structure to them in the form of syntax trees which contain nonterminals as well as terminals. For example, the following is a syntax tree for "Hall discovered Phobos":



CFGs have the advantage of simplicity, and they can also be readily extended to accommodate semantic processing either by annotating the syntax trees with semantic values and then evaluating those trees, or by associating semantic functions directly with the production rules and applying those functions during the parsing process, resulting in what are commonly called syntax-directed evaluators.

However, CFGs also have limitations with respect to the explanation and analysis of natural language. For example, in order to accommodate agreement (such that the expressions "Phobos spins" and "Phobos and Deimos spin" are admitted, but "Phobos spin" is rejected), we would have to subdivide the nonterminal `termphrase` into `singular-termphrase` and `plural-termphrase`, subdivide the nonterminal `verbphrase` likewise (and further subdivide these and other categories to accommodate other forms of agreement), and then replace the single rule, that a sentence consists of a `termphrase` followed by a `verbphrase` by a set of rules which admits only those combinations of subdivisions of categories that agree. The grammar would become even larger in order to accommodate long distance agreement which is necessary to admit "Which moons did you say were discovered by Hall?" but reject "Which moon did you say were discovered by Hall?". Similarly, in order to deal with the fact that transitive verbs have different numbers of arguments (e.g., "Hall discovered Phobos" and "Hall gave Kuiper a telescope"), we would have to subdivide the nonterminal `transverb`. This would have a multiplicative effect on the size, resulting in huge grammars.

Context-free grammar has also been criticized for its independence from semantics in the sense that the syntactic categories can be chosen independently of semantic concerns leading to difficulties when semantic values and evaluation functions are associated with the production rules.

An additional difficulty with CFG is that it is widely, though not unanimously, accepted that natural language is not context free. Various sublanguages have been

identified which, it is claimed, cannot be generated by CFG. For example,

—the language of reduplicated expressions of the form awa, where a and w are sequences of words, for example, "a state within a state" and "a church within a church", etc.

—the language of multiple-agreement expressions or counting dependencies of the form $a^n b^n c^n$, such as "John, Sue and James, a widower, widow and widower, subsequently married Paula, Christopher, and Isabelle".

—the language of cross-serial dependencies of the form $xa^m b^n yc^m d^n z$ found in Dutch and Swiss German, such as, "Jan sait das mer d'chind em Hans es huus haend wele laa halfe aastriiche". (Jan said that we wanted to let the children help Hans paint the house.).

More comprehensive discussion of the non-context-freeness of natural language can be found in [Shieber 1985], Savitch [1989], and Kudlek et al. [2003] from which the cited examples were derived.

The noncontext-free sublanguages discussed can be generated by context-sensitive grammar, which is similar to CFG except that production rules are of the form: xAy ::= xay, where A is a single nonterminal, x and y are (possibly empty) strings of terminals and nonterminals, and a is a nonempty string of terminals and nonterminals. The name context-sensitive comes from the fact that the context defined by x and y determines that A can be replaced by a.

Context-sensitive grammar, however, has two shortcomings with respect to natural language analysis: 1) it is too expressive in the sense that it can also generate sublanguages that do not occur in natural language, and 2) all known algorithms for parsing context-sensitive languages have exponential time dependency. Consequently, the notion of mildly context-sensitive grammar has been developed. This grammar is only slightly more powerful than CFG; there is a limit on the depth of instantiations of cross-serial dependencies; and the recognition problem is solvable in polynomial time. Mildly context-sensitive grammar also captures many context-free linguistic features more succinctly than CFG.

In the following, we include some comparison of grammars with each other and with context-free and context-sensitive grammar. Two grammars are said to be weakly equivalent if they generate the same language. Two grammars are strongly equivalent if they assign the same syntax trees to their sentences (ignoring differences in the identifiers used to denote the nonterminals in the two grammars).

3.2. Categorial Grammar

One of the first formal approaches to linguistics, developed by Ajdukiewicz [1935] and Bar-Hillel [1953], is based on the notion that linguistic structures can be complete or incomplete and that grammatical composition is the process of completing incomplete structures. For example, in the sentence "Phobos spins", the proper noun "Phobos" might be deemed to have a simple complete meaning, that is, the moon Phobos. However, the verb "spins" might be deemed to be incomplete in that its meaning is part of a proposition which requires a subject to become complete. Categorial Grammar (CG) is based on this notion and considers syntactic constituents as functions which combine with each other to form composite structures. Expressions are assigned categories which specify how they combine with expressions of other categories to create larger expressions. Analysis of an expression involves the application of inference rules to the categories assigned to its component parts in order to determine the category of the whole expression. In CG, the set CAT of categories is defined as follows, where B is the

set of basic categories:

```
if X ∈ B then X ∈ CAT
if X,Y ∈ CAT then X/Y,X\Y ∈ CAT        there are no other categories
```

In basic CG, categories are combined using two rules, one for right and one for left function application:

```
X/Y, Y =>right X              Y, Y\X =>left  X
```

For example, given `B = {S, N, T}` denoting the categories of sentence, noun, and termphrase respectively, we can define the following lexicon:

Hall, Kuiper, Phobos, Deimos	∈ T		
moon	∈ N	*spins*	∈ T\S
every	∈ T/N	*discovered*	∈ (T\S)/T

We can now use these categories in the analysis of expressions. For example, to show that "every moon spins" and "Hall discovered Phobos" are sentences:

```
every       moon      spins            Hall    discovered    Phobos
-----       ----      -----            ----    ----------    ------
 T/N          N        T\S              T        (T\S)/T       T
 --------=>right                                 ---------=>right
      T                T\S              T              T\S
      ---------------=>left              ------------=>left
                S                                S
```

Note that in Categorial Grammar, the rules are written as accepting rules, indicating how smaller components can be combined to create larger expressions, unlike context-free grammar where the rules are written as producing rules that, when used from left-to-right, generate expressions of the language. As such, CG is lexicalized in that only a small number of rules of composition are used, and most of the syntactic features of the language are derived from syntactic features of individual words in the lexicon.

Lambek [1958] formalized the concept of syntactic categories by defining a calculus consisting of a set of axioms and rules which can be used to deduce the category to which an expression belongs. In Lambek calculus, \ and / are treated as forms of logical implication.

One of the advantages claimed for Categorial Grammar is the ease with which compositional semantic theories can be associated with it. The assignment of categories to words in CG is strongly motivated by semantic considerations. The category not only determines the syntactic property of the word, it usually determines the semantic type of the word's denotation. Given an appropriate formalism for representing semantic values (e.g., the typed lambda calculus), the rules for semantic composition follow directly from the rules for syntactic composition, thereby complying with the principle of rule-to-rule correspondence and contributing to the compositionality of the approach.

In 1960, Bar-Hillel [1953] proved that basic CG is weakly equivalent to CFG, and a similar proof was given for the Lambek calculus. These discoveries led to a wane in interest in these approaches. However, interest was reawakened in the mid 70s after Montague [1970, 1973] developed a type-based approach to semantics and associated it with a grammar that was similar to CG (see Section 3.3).

In addition to the fact that basic Categorial Grammar is weakly equivalent to CFG, it has other shortcomings as discussed in Baldridge and Kruijff [2004] who discuss the following phrases: team that defeated Germany, team that Brazil defeated, the team that I thought that you said defeated Germany easily yesterday. Words in these phrases would have to be assigned several different categories to provide a categorial analysis. The resulting categorial ambiguity causes the grammars to become unwieldy.

Basic CG has been extended in various ways to overcome its shortcomings. One approach is to add more rules of composition, resulting in Combinatorial Categorial Grammar discussed in Section 3.4. Another approach was to develop the Lambek calculus resulting in, for example, categorial type logic [Moortgat 1997] and type-theoretic grammar [Ranta 1994] discussed in Section 3.6.

3.3. Montague Grammar

One of the most influential approaches to natural language interpretation was developed in the late sixties and early seventies by Richard Montague. That work is described in a number of densely-packed papers, for example, Montague [1970, 1973] and Thomason [1974], in more accessible form in a comprehensive paper written by Partee [1975] and in a book by Dowty et al. [1981]. An early collection of papers written in the Montague framework is given in Partee [1976]. Historical overviews of Montague's approach, which compare it with other linguistic theories, are available in Partee and Hendricks [1997] and Partee [2001].

The following is a brief and highly-limited introduction to some of Montague's ideas. In particular, we will say nothing about modal or intensional aspects of natural language, as these topics require substantial background discussion. More complete descriptions can be found in the references just given and in the numerous papers written by Partee and other linguists who have contributed to the development of Montague-like compositional theories.

Central to Montague's approach is his claim that there is no intrinsic difference between natural and formal languages and that natural language can be described in terms of a formal syntax and an associated compositional semantics. The relationship between the syntax and semantics is similar to that in the denotational semantics approach to the formal specification of programming languages with the exception that expressions of natural language have first to be disambiguated before interpretation. Such disambiguation involves mapping natural language expressions to one or more unambiguous expressions in a syntactic algebra. These expressions are then mapped to expressions in a semantic algebra through a homomorphism.

Montague Grammar is similar to Categorial Grammar in some respects: categorial-like names are used for syntactic categories, and semantic types are similar to those in CG. It should be noted, however, that Montague Grammar is not strictly a Categorial Grammar as it includes transformation rules to move, delete, and substitute syntactic components. In Montague Grammar, each disambiguated syntactic expression denotes a function in a function space constructed over a set of entities, the Boolean values true and false, and a set of states each of which is a pair consisting of a possible world and a point in time. The function space is described using the notation of lambda calculus. Each syntactic category is associated with a single semantic type. Each syntax rule, which shows how composite expressions in a category are created from their syntactic constituents, is associated with a semantic rule, which shows how the meanings of those composite expressions are computed from the meanings of their components. The primary rule for syntactic composition is juxtaposition (phrases being placed next to each other). The primary rule for semantic composition is function application.

In Montague's Proper Treatment of Quantification, (PTQ) he developed a specific syntax and semantics for a fragment of English [Montague 1973]. In PTQ (ignoring intensional aspects which involve states), nouns such as "planet" and intransitive verbs such as "spins" denote predicates over the set of entities, that is, characteristic functions of type `Entity -> Bool`, where `x -> y` denotes the type of functions whose input is a value of type `x` and whose output is of type `y`. Proper nouns do not denote entities directly. Rather, they denote functions defined in terms of those entities. For example, the proper noun "Mars" denotes the function `λp p Mars` where `Mars` denotes the entity Mars. According to the rules proposed by Montague, the phrase "Mars spins" is interpreted as follows, where `x => y` indicates that `y` is the result of evaluating `x`. Note that the denotation of words such as "spins", is given in nonitalic computer font. For example, `spins_p`, is shorthand for "the predicate associated with the word spins":

```
(λp p Mars) spins_p => spins_p Mars
```

Quantifiers (also called determiners) such as "every", and "a" denote higher-order functions of type `(Entity -> Bool) -> (Entity -> Bool) -> Bool`. For example, the quantifier "every" denotes the function:

```
λpλq ∀x (p x) → (q x)
```

where → is logical implication. Accordingly, the phrase "every planet spins" is interpreted as follows:

```
(λpλq ∀x p(x)→q(x)) planet_p spins_p
=>(λq ∀x planet_p(x)→q(x)) spins_p
=> ∀x planet_p(x)→spins_p(x)
```

Constructs of the same syntactic category denote functions of the same semantic type. For example, the phrases "Mars" and "every planet", which are of the same syntactic category, both denote functions of type `(Entity -> Bool) -> Bool`. Montague's approach is highly orthogonal; many words that appear in differing syntactic contexts denote a single polymorphic function thereby avoiding the need to assign different meanings in these different contexts. For example, the word "and", which can be used to conjoin nouns, verbs, term-phrases, etc., denotes the polymorphic function: `λgλfλx (g x) & (f x)`. Using these denotations, the phrase "Phobos and Deimos spin" is interpreted as follows:

```
=> (and (Phobos Deimos)) spins_p
=> ((λgλfλx (g x) & (f x)) (λp p Phobos) (λp p Deimos)) spins_p
=> ((λx ((λp p Phobos) x) & ((λp p Deimos) x))) spins_p
=> ((λp p Phobos) spins_p) & ((λp p Deimos) spins_p)
=> (spins_p Phobos) & (spins_p Deimos)
=>               True
```

Montague's semantics for transitive verbs is somewhat complex. This appears to be a consequence of the fact that, although he defined the denotation of proper nouns as, for example, `λp p Phobos`, he viewed these denotations and the denotations of other term-phrases as being of type `(Entity -> Bool) -> Bool`. This creates a difficulty when attempting to define a denotation for transitive verbs which can work in phrases such as"Hall (discovered Phobos)" (note that we use brackets to illustrate the order in which the functional denotations would be applied). The denotation of "discovered" would

have to be of type:`((Entity ->Bool) ->Bool) -> (Entity -> Bool)` in order to accept the Montague-typed denotation of "Phobos" as input and return an appropriately-typed value for input to the Montague-typed denotation of "Hall". Montague appears not to have developed such a denotation. Instead, he used an approach in which transitive verbs are left uninterpreted, while the phrase in which they appear is converted to an intermediate form, at which point a somewhat convoluted syntactic manipulation takes place converting the expression to a form which involves the binary predicate associated with the verb. An example of this complex process can be found in Dowty et al. [1981, p. 216]. Frost and Launchbury [1989] and Frost [2006] have noted, however, that the type of denotations of proper nouns, such as `λq q Phobos` is not: `(Entity -> Bool) -> Bool`, but: `(Entity -> a) -> a`, where `a` denotes any type. From this, we can derive a direct denotation of transitive verbs by working backwards from the required result. For example, according to Montague, the denotation of "Hall (discovered Phobos)" should be `discover_pred(Hall, Phobos)`. Therefore:

```
(λq q Hall) (discover (λp p Phobos)) => discover_pred(Hall, Phobos)
```

one solution of which is `discover (λp p Phobos) = λx discover_pred(x, Phobos)` and one solution to this is `discover = λz(z λxλy discover_pred(y,x))`. The following is an example application of this denotation in interpreting the sentence "Hall discovered Phobos" illustrating the polymorphic type of the denotations of "Hall" and "Phobos":

```
(λp p Hall) ((λz z(λxλy discover_pred(y,x))) (λq q Phobos))
=> (λp p Hall)((λq q Phobos)(λxλy discover_pred(y,x)))
=> (λp p Hall) ((λxλy discover_pred(y,x)) Phobos)
=> (λp p Hall) (λy discover_pred(y,Phobos))
=> (λy discover_pred(y,Phobos)) Hall
=> discover_pred(Hall,Phobos)
```

Although this denotation for transitive verbs is not well known, it has been suggested by others, especially in the logic programming community, for example, Blackburn and Bos [2005] who attribute it to Robin Cooper at the University of Göteborg. Also, in a personal communication, Barbara Partee, who is arguably the foremost authority on Montague Semantics, has pointed out that, although the previous treatment of transitive verbs is not standard in linguistics, Angelika Kratzer, at the University of Massachusetts, has done something similar under the label of "argument identification", and that Hendricks [1993] has proposed type-lifting to achieve a similar result in a comprehensive analysis of categories and types.

The reason for including the aforementioned account is that it exemplifies the use of a well-known functional-programming tactic to easily and systematically develop a solution which might otherwise not be immediately apparent. As such it is the first of several examples discussed in this survey of the application of LFP to NLI.

In addition to the direct interpretation of natural language summarized in this section, Montague also defined an indirect approach in which disambiguated expressions of natural language are translated into expressions of a higher-order modal intensional logic called IL. Montague claimed that the use of IL as an intermediate form is dispensable and serves only to help explain the relationship between syntax and semantics. However, one of the criticisms of Montague's theory is that it cannot explain some linguistic features without recourse to analysis of the structure of the IL intermediate representation. For example, Pereira [1990] has noted that Montague grammar needs to invoke constraints on intermediate intensional logical forms to explain why sentences such as "A woman who saw every man disliked him" are ungrammatical, and

why, in sentences such as "Every man saw a friend of his", the "every" phrase has a wider scope than the "a" phrase. Pereira states that such reliance on the logical form rather than the semantic interpretation goes against the spirit of compositionality, and that it also belies the notion that the intermediate IL representation was a dispensable part of Montague's framework.

The major advantages of Montague's approach are the homomorphism from syntax to semantics, the orthogonality of the semantic values and rules, and the resulting compositionality. During the 70's, Montague's approach was slowly accepted by the linguistic community, largely owing to researchers such as Partee [1975, 1976] and Dowty [1979] who, with others, extended the approach to accommodate a wider range of linguistic features.

3.4. Combinatory Categorial Grammar

Combinatory Categorial Grammar (CCG) is a form of Categorial Grammar in which the two basic rules of right and left function application are augmented with additional rules for combining categories [Steedman 1991, 2004; Steedman and Baldridge 2003].

Pure Categorial Grammar is weakly equivalent to context-free grammar, whereas CCG is mildly context sensitive. The rules of CCG correspond to the combinators identified by Curry and Feys [1958], and hence its name. The additional rules include a rule of coordination: `X conj X =>conj X`, a rule of forward composition: `X/Y Y/Z =>compose X/Z`, and a rule of subject-type raising: `T =>raise S/(T\S)`, an example of which is:

```
Hall    discovered   and  Kuiper   discovered

 T        (T\S)/T    conj   T        (T\S)/T
=>raise                     =>raise
S/(T\S)                     S/(T\S)
--------=>compose           ---------=>compose
     S/T                          S/T
     ---------------------------=>conj
                  S/T
```

Use of the rules of CCG are constrained by three principles called *adjacency*, *consistency*, and *inheritance*. It is claimed that the rules and principles not only provide an explanation for many features of English (and Dutch) but that they also capture certain features that are common to all natural languages.

As in Montague, each grammatical category is associated with a single semantic type, and the semantic values are functions represented as lambda terms. A principle of, *combinatory transparency* determines how semantic values are computed. This principle states that the semantic interpretation of a category that is created using one of the rules is determined by interpreting "/" and "\" as mappings between two sets. For example, the following is the rule of *forward composition* with semantics, where `C:s` indicates that `s` is the semantic value of the phrase `C`:

```
X/Y:f Y/Z:g => X/Z:λx f(g x)
```

One advantage claimed for CCG [Steedman 1994] is that it is easy to relate the grammar to a compositional semantics by assigning semantic values to the lexical entries and semantic functions to the combinatory rules such that no intermediate representation is required. Steedman [1999] has also shown how quantifier-scope ambiguities

can be accommodated in this framework without recourse to rules for changing the structure of an intermediate representation.

3.5. Type-Logical Grammar

During the 1980s, a deductive form of Categorial Grammar was developed by van Benthem [1986] and Moortgat [1987]. This approach began with the Lambek calculus. Subsequently, van Benthem [1987; 1991] extended the Lambek calculus with a compositional semantics, using simply-typed lambda terms to represent formulas of predicate calculus. According to the Curry-Howard isomorphism [Girard et al. 1988], simply-typed lambda terms are proofs in intuitionistic logic which embeds the Lambek calculus. Van Benthem used this correspondence to establish a relationship between the Lambek calculus and Montague semantics. Moortgat [1988; 1990] also investigated the relationship of the Lambek calculus to logical semantics and discussed Montague's theory from this perspective. A review of research on logical aspects of computational linguistics up to the mid 1990s is given in Blackburn et al. [1997].

3.6. Type-Theoretical Grammar

In CFG and CG, the rules for creating expressions from their constituents depend only on their course-grained syntactic categories. For example, consider the following, where `S` is the category of sentences, `PN` the category of proper nouns, `VP` the category of verb phrases, and `TV` the category of transitive verbs:

```
S  ::= PN VP          PN ::= Hall | Phobos
VP ::= TV PN          TV ::= discovered
```

Both of the sentences "Hall discovered Phobos" and "Phobos discovered Hall" can be derived, even though the latter could be thought of as being ill-typed in the sense that moons cannot discover anything. In order to deal with this (and also to accommodate other forms of agreement), type-theoretic grammar was developed to place constraints on the domains of categories at the level of abstract syntax [Ranta 1994]. This process can be thought of as adding more fine-grained syntax to context-free and Categorial Grammar without having to subdivide rules and substantially increase the size of the grammar.

The first step is to make explicit the structures which are being created by application of the derivation rules. For the concrete syntax, the rules are rewritten as follows, where `x::X` indicates that `x` is of category `X`, and `x ++ y` is the string obtained by appending `x` to the front of `y`.

```
a:: PN     b::VP         c:: TV  d:: PN
----------------         --------------
  a ++ b :: S             c ++ d :: VP
```

Rules for abstract syntax are formulated in a similar way. For example.

```
a:: PN     b::VP         c:: TV  d:: PN
----------------         --------------
SUBJ(a,b) :: S           OBJ(c,d) :: VP
```

where the `SUBJ` and `OBJ` are value constructors. The abstract syntax for "Hall discovered Phobos" is the tree: `SUBJ(Hall, OBJ(discovered, Phobos))`

Concrete representations can be obtained from the abstract syntax through a process of linearization which applies rules such as

```
lin Hall      = "Hall"        lin SUBJ(x,y) = lin x ++ " " ++ lin y
```

The next steps are to subdivide the basic categories in the abstract syntax to reflect the different semantic domains of their denotations, and then to define a type hierarchy on those subdivisions:

```
PN(humans)        ::= Hall | etc.          PN(moons)::= Phobos | etc.
TV(humans,things)::= discovered | etc.         moons ⊆ things, etc.
```

The rules of the abstract grammar are then modified to make explicit the requirement for type matching. For example, `S ::= PN(A) VP(A)` meaning that the types of the proper noun and the verb phrase must be compatible. Another example is `VP(A) ::= TV(A,B) PN(B)`, meaning that a verb phrase of type `A` is constructed from a transitive verb with a subject of type `A` and an object of type `B`, and a proper noun whose type is compatible with `B`.

Note that not all type constraints can be defined in context-free rules. However, due to the fact that the abstract syntax trees are terms of a formal type theory, context-sensitive type-checking of these terms can be used to enforce a wide range of constraints.

3.7. The Differing Roles of Linguistic Theories in NL Explanation and NLI Development

The inability of a linguistic theory to explain all features of natural language is important from a linguistic perspective but less so with respect to its use in implementing NLIs. There are three reasons for this.

—The state-of-the-art in NLI is far behind that of linguistics in terms of the range of expressions that can be accommodated. The full power of existing theories of language has yet to be employed. For example, there would appear to be no NLI which can accommodate intensionality.

—The efficiency with which a theory can be implemented is irrelevant from a linguistic perspective but is of importance in the creation of NLIs. In many applications, some reduction in expressibility may be acceptable for an appropriate improvement in response time.

—The linguistic concern that a theory admits expressions that do not occur in a natural language and is therefore faulty in its explanation of that language, is of less importance in NLI. It usually does not matter if the system can accommodate expressions that are grammatically ill formed provided that the response is sensible. In fact, such robustness is considered to be an advantage in many applications.

Consequently, theories that have shortcomings from a linguistic point of view may still be of interest to those who are building NLIs. We shall see throughout this survey that much of the research on the use of LFP in NLIs has been based on relatively-simple subsets of the theories previously discussed, and that there remains much to be done before the full potential of those theories can be exploited.

4. LAZY FUNCTIONAL PROGRAMMING

In the introduction, we gave a brief definition of what lazy functional programming is. In this section, we provide a short introduction to some LFP languages, the notation that

we will use throughout this survey, and a discussion of the features of LFP. Readers who are familiar with lazy functional programming can skip this section and move directly to Section 5.

4.1. Examples of LFP Languages

Miranda[1] [Turner 1979; 1985; 1986] was one of the earliest lazy functional programming languages to be relatively widely used, especially for teaching. Miranda has had an important influence on the development of other LFP languages. More information on Miranda is available from `http://miranda.org.uk`

LML is a lazy variant of the functional programming language ML. LML was developed by Johnson and Augustsson at Chalmers University around 1984 and was used to implement the first Haskell compiler.

Id [Nikhil 1993] is a lazy functional dataflow programming language designed in the '80s for execution in a parallel computing environment.

Hope is a pure functional programming language that was also developed in the mid '80s. It was one of the first programming languages to use call-by pattern. The original version used eager evaluation, but there are versions with lazy lists and lazy constructors, see `http://www.soi.city.ac.uk/ ross/Hope/`

Clean is a lazy functional language which was first described by Brus et al. [1987]. The current version of Clean uses *uniqueness typing* to allow destructive update of data structures and, arguably, a more natural interface between declarative functional programs and the imperative environments in which they execute. This feature facilitates the use of Clean in the development of window-based and distributed applications. More information on Clean is available from `http://www.cs.ru.nl/ clean/`

Haskell is the product of a committee that was established in 1987 with the objective of creating a standard LFP language. The first version, Haskell 1.0, was defined in the late eighties and is described in Hudak et al. [1992]. The latest version, Haskell 98, is described in detail in Peyton-Jones [2003]. The current Haskell report, together with links to resources and descriptions of applications built in the language, can be obtained from `http://www.haskell.org/`

Haskell has replaced Miranda as a teaching language at many sites due to the fact that it is freely available, has been ported to many platforms, and has good technical support.

4.2. The Notation of Haskell

We use the following subset of Haskell in our examples.

(1) `f = e` defines `f` to be a function which returns the value of the expression `e`.

(2) The notation for function application is simply juxtaposition as in `f x`. Function application has higher precedence than any operator.

(3) Function application is left associative. For example, `f x y` is parsed as `(f x) y`, meaning that the result of applying `f` to `x` is a function which is then applied to `y`. Round brackets are used to override the left-associative order of function application. For example, `f (x y)`.

(4) `f a1 ... an = e` can be read as defining `f` to be a function of `n` arguments whose value is the expression `e`. However, in higher-order languages, functions can be passed as parameters and returned as results. Every function of two or more arguments is actually a higher-order function, and the correct reading of `f a1 ... an`

[1]Miranda is a trademark of Research Software Ltd.

`= e` is that it defines `f` to be a higher-order function which, when partially applied to input `i`, returns a function `f'` such that `f' a2 ... an = e'`, where `e'` is `e` with the substitution of `i` for `a1`. For example, consider the function `add` defined as follows: `add x y = x + y`. The function `incr`, which adds one to its input, can be defined in terms of the partial application of `add` as follows: `incr = add 1`, such that `incr 4 => 5`.

(5) `x 'f' y` allows `f` to be used as an infix operator.

(6) Functions can be composed with the dot operator, for example, `(f . g) x = f (g x)`

(7) In a function definition, the applicable equation is chosen through pattern matching on the left-hand side in order from top-to-bottom together with the use of guards, for instance, in a long form of the definition of the absolute function:

```
abs  0              = 0
abs  n |  n  > 0    = n
       | otherwise  = -n
```

(8) Round brackets with commas are used to create tuples, for example, `(x, y)` is a binary tuple. Square brackets and commas are used to create lists, for instances, `[x, y, z]`. The empty list is denoted by `[]`, and `x : y` denotes the list obtained by adding the element `x` to the front of the list `y`. Lists are apended with ++. The representation of strings enclosed in double quotes is shorthand for lists of characters, such as, `"abc" = ['a','b','c']`.

(9) Lists can also be created using the list-comprehension construct which has the general form `[values | generators, conditions]` For example,

```
[x^2 | x <- [1..10], odd x] => [1, 9, 25, 49, 81]
```

(10) `T1 -> T2` is the type of functions with input type `T1` and output type `T2`. The declaration `f :: T` states that `f` is of type `T`, and `T1 = T2` declares `T1` and `T2` to be type synonyms.

(11) New types can be defined using the `data` construct. For example `data Colour = Red | Blue | Green` introduces the user-defined type `Color` and three nullary constructors.

(12) Haskell supports parametric polymorphic types which involve type variables that are universally quantified over all types. Type variables begin with an uncapitalized letter to distinguish them from specific types such as `Integer`. For example, consider the function `length` which returns the length of lists of elements of various types. Its type is `[a] -> Integer`, where `a` is a type variable.

4.3. Features of LFP Languages

In the introduction, we defined LFP languages by describing those programming constructs which they do not have. However, as John Hughes [1989] has elegantly argued, it is not what lazy functional programming languages lack that gives them their real power, it is what they have. In the following, we summarize some of the advantages of LFP.

—A lazy functional program is declarative in the sense that it consists of a set of definitions which can be presented in any order, simplifying program development and allowing equational reasoning to be used in program analysis and transformation to more efficient forms.

—Higher-order functions can be defined which take functions as arguments and/or return functions as results. Higher-order functions can be used to capture frequently

used patterns of computation. For example, suppose that we start with the function `sumlist` which is not higher-order:

```
sumlist   []   = 0
sumlist (n:ns) = n + sumlist ns
```

An example application is `sumlist [5,8,2] => 15`. The structure of this program can be abstracted out and encapsulated in the higher-order function `foldr` defined as shown below. Now, `sumlist, productlist`, and other list processing functions can be defined by partial application of `foldr` to two of its arguments:

```
foldr op unit []     = unit
foldr op unit (n:ns)
  = n `op` foldr op unit ns

sumlist     = foldr (+) 0
productlist = foldr (*) 1
concatlist  = foldr (++) []
```

This ability to define higher-order functions enables a new kind of modularity which is more difficult to achieve in other programming languages.

—Higher-order functions can be defined as infix operators and partially applied to their arguments, allowing programs to be built with structures (form) that closely follows the specification of what they are intended to do (their function). These higher-order infix operators are often referred to as combinators. A well-established example of this is the use of parser combinators to build parsers whose structures are closely related to the grammars defining the languages to be parsed. For example, a parser for a possibly empty sequence of adjectives can be defined as follows, where the combinators `term, orelse`, and `then1` have been appropriately defined.

```
adj  = (term "red") `orelse`  (term "blue")  `orelse` ...
adjs = empty `orelse`  (adj `then1` adjs)
```

The identifier `then1` is used due to the fact that `then` is a reserved word in Haskell. We discuss the definition and use of parser combinators in detail in Section 5.

—LFP languages are strongly typed meaning that all values have a type, and the language prevents the programmer from applying a function to a value of the wrong type. It is impossible to inadvertently convert a value of one type to another type.

—LFP languages are statically typed. This means that the language infers types automatically, checks for type mismatch, and catches type errors at compile time.

—LFP languages are polymorphic and automatically infer the most general type. For example, the type of `foldr` is inferred to be: `(a->b->b)->b->[a]-> b`, which can be read as follows: `foldr` takes as argument a binary operator, whose input values can be of any types `a` and `b` and whose output value is of type `b`, and returns as its result a function `f'` which takes a value of type `b` as input and which returns as its result a function `f''` which takes a list of values of type `a` as input, and which returns as its result a value of type `b`. The type of `foldr` returned by the type system indicates that the function is more general than suggested by the examples given earlier of its use in defining `sumlist, productlist`, and `concatlist` (in each of which `a` and `b` are of the same type). The inferred type of `foldr` shows that it can also be used to define functions where the input types of its first operator argument are different. For example, the following is a definition of a function `reverse` which reverses the order of elements in a list given as argument:

```
reverse = foldr put_on_end []   where  put_on_end x r = r ++ [x]
```

—Lazy evaluation does not compute arguments to functions until, and unless, they are required. This provides a form of modularity that is difficult to achieve in other types

of programming: the ability to separate the generation and use of data structures that are potentially infinite in size. For example, consider the following program which returns the squares of the first three natural numbers (assuming that the function `take_3` has been defined elsewhere):

```
first_three_squares = take_3 [x^2 | x <- [1..]]
```

The comprehension following `take_3` is a reusable component that generates the infinite list of squares.

—It should be noted that Haskell does not have any built-in support for unification as does the logic programming language Prolog. However, van Eijck [personal communication] notes that unification is easy to implement in Haskell and gives an example in his forthcoming book *Computational Semantics with Type Theory* `http://www.cwi.nl/ jve/cs/`.

4.4. A Note on Haskell Types and Classes

Haskell augments parametric polymorphic typing with ad hoc polymorphism. We explain this feature in the following using an example derived from Hudak et al. [2000].

With parametric polymorphism, when a function's type is defined using an expression involving a type variable `a`, the `a` is intended to mean any type. However, there are situations where the type should be restricted. For example, in

```
equal_fsts:: [a]  ->  [a]   ->  Bool
equal_fsts  (x:xs)   (y:ys) = x == y
equal_fsts    n        m    = False
```

The type declaration states that the function can be applied to two lists whose elements can be of any type `a`, provided they are of the same type. However, the function is really only applicable to lists containing elements which can be tested for equality using `==`. This is not the case for functions for which the equality test is, in general, computationally intractable, and an attempt to apply `equal_fsts` to lists of functions would cause a runtime error. In Haskell, this problem is addressed through the introduction of an additional kind of polymorphism, called ad hoc polymorphism, in which type classes are defined together with associated overloaded operators. For example,

```
class Eq a where (==) :: a -> a -> Bool
```

This definition states that, if a type `a` is an instance of the class `Eq`, then there is an operator `(==)` of type `a -> a -> Bool` associated with it. The definition can also be thought of as constraining the type of `==` as follows, where `(Eq a)` is called the context in which `==` has type `a -> a -> Bool`:

```
(==) :: (Eq a) => a -> a -> Bool
```

(Note the different meaning of `=>` in this context). Now we can define instances of the class `Eq`, together with associated behaviors for `==`, as, for example, in the following, where `integerEq` and `floatEq` are built-in Haskell operators.

```
instance Eq Integer where  x == y  =  x `integerEq` y
instance Eq Float   where  x == y  =  x `FloatEq`   y
```

Now we can constrain the type of `equal_fsts` to apply only to lists containing elements for which `==` has been defined: `equal_fsts::(Eq a)=>[a]->[a]->Bool`

Classes can be extended as, for example, in the following where the class `Eq` is said to be a superclass of `Ord`:

```
class (Eq a)  =>  Ord a     where   (),(=),(>=),(>):: a -> a -> Bool
                                     max, min      :: a -> a -> a
```

The advantages of this are that, the context `(Ord a)` can now be used in type declarations rather than `(Eq a, Ord a)`, and operations from the superclasses `Eq` can be used in definitions of instances of the subclass `Ord`.

New classes may also be defined in terms of more than one superclass as in the following that creates a class `New` that inherits operations from `Eq` and `Conv` (assumed to have been defined elsewhere): `class (Eq a,Conv a) => New a`.

4.5. Monads

According to Wadler [1990], *monads* were introduced to computing science by Moggi [1989] who noticed that reasoning about pure functional programs which involve handling of state, exceptions, I/O, or nondeterminism can be simplified if these features are expressed using monads. Inspired by Moggi's ideas, Wadler proposed monads as a way of systematically adding such features to algorithms. The main idea behind monads is to distinguish between the type of values and the type of computations that deliver those values.

A monad is a triple `(M, unit, bind)` where `M` is a type constructor, and `unit` and `bind` are polymorphic functions. `M` is a function on types that maps the type of values into the type of computations producing those values. `unit` is a function that takes a value and returns a corresponding computation. The type of `unit` is `a -> M a`. The function `bind` enables sequencing of two computations where the value returned by the first computation is made available to the second (and possibly subsequent) computation. The type of `bind` is `M a -> (a -> M b) -> M b`

In order to use monads to provide a structured method for adding a new computational feature to a functional program, we begin by identifying all functions that will be involved in the new feature. We then replace those functions, which can be of any type `a -> b`, by functions of type `a -> M b`. Function applications are then replaced by the function `bind` which is used to apply a function of type `a -> M b` to a computation of type `M a`. Those values which are not involved in the new feature are converted by applying `unit` to them into computations that return the value but do not contribute to the new feature. As an example, discussed further in Section 5.4, this process can be used to add a memo table to top-down parsers implemented as purely functional programs. The memo table is threaded through all component parser applications and allows results to be reused if the parser is ever reapplied to the same input. This reduces time complexity for recognition from exponential to polynomial for highly ambiguous grammars and also allows left recursive grammars to be directly implemented as modular top-down parsers. Another use of monads, discussed in Section 8.3, is to systematically extend theories of natural language to accommodate additional linguistic features.

A more complete account of monads can be found in Wadler [1990; 1995] and in the tutorial by Hudak et al. [2000].

5. USE OF LFP IN SYNTACTIC ANALYSIS

Later in the article, we discuss the use of LFP in systems which integrate the syntactic and semantic analysis of natural language. However, before we discuss those systems, we review the use of LFP in syntactic and semantic analysis separately in this and the next section, respectively.

5.1. Summary of Techniques for Language Recognition and Parsing

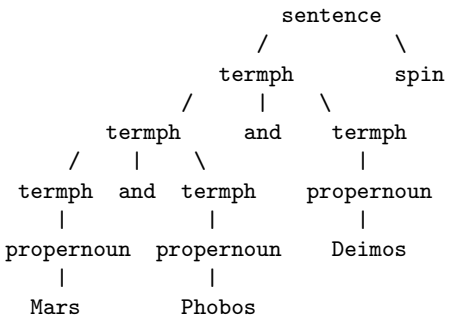
We begin with a brief overview of techniques for the automatic recognition and parsing of languages. Readers who are familiar with this material can skip this section and move directly to Section 5.2.

In Section 3, we defined a language as the set of sentences that can be generated from a grammar using the production rules as left-to-right rewrite rules. It follows, therefore, that if we are given a sequence of terminals, we can analyze that sequence with respect to a given grammar. One question that we could ask is whether or not the sequence belongs to the language defined by the grammar. This form of syntactic analysis is called recognition and is decidable for context-sensitive and context-free grammars. The second form of syntactic analysis involves showing how a sentence might be derived from a given grammar. This involves providing a structure which links the terminals of the sentence to the start symbol through reference to the nonterminals involved in the derivation. This form of analysis is called parsing.

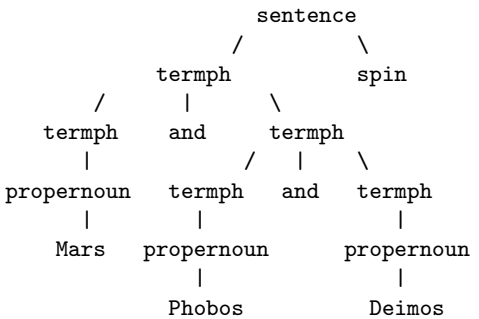
The structure produced by a parser is often represented in what is called a syntax tree or parse tree, as illustrated in Section 3.1. In some cases, more than one syntax tree can be generated for a single sentence with respect to a single grammar. For example, consider the following simple grammar, where `termph` is an abbreviation for `termphrase`:

```
sentence    ::= termph spin
termph      ::= propernoun | termph and termph
propernoun  ::= Mars | Phobos | Deimos
```

The sentence “Mars and Phobos and Deimos spin” has two syntax trees: t



and



The first tree corresponds to a left-most derivation of the sentence and the second tree corresponds to a right-most derivation of the sentence. Left-most and right-most

refer to the order in which the nonterminals on the right-hand side of a production would expanded to generate the sentence.

Parsers differ in many ways as shown by the following examples.

—They can differ in the direction in which the tree is built. They can build the tree from the top down, reaching to the terminals in the input sequence by creating a tree with the start symbol at the top and then recursively predicting which right-hand sides of rules to apply and by expanding nonterminal nodes in the tree appropriately. Alternatively, they can build the tree from the bottom up by shifting input terminals into a workspace, and then reducing sequences of terminals and nonterminals on the fringe of the tree to nonterminals at a higher level in the tree. Efficient bottom-up parsers make use of a table which is generated from the grammar before parsing commences. The table contains information which is used by the parser to determine whether to shift or reduce, and on reduce, which rule to use. Some parsers use a combination of top-down and bottom-up techniques.

—They can use a *depth-first* strategy in which a complete branch is constructed from the start symbol to a terminal (in top-down construction) or from a terminal to the start symbol (in bottom-up). Alternatively, they can use a *breadth-first* strategy in which all nodes at a particular level are constructed before any node in the next (lower level in top-down, and upper level in bottom-up) are built. Or they can use a combination of depth-first and breadth-first.

—They can differ in the order in which terminal symbols are attached to the tree. In *nondirectional* methods, the terminals are attached in an arbitrary order. In *left- and right-directional* methods, the terminals are attached to the tree in the order or reverse order in which they appear in the input sequence.

—They can differ in the order in which the nonterminals on the right-hand side of a production are expanded in the tree. For instance, the first tree in the previous example could be produced by a parsing strategy which first expands the left-most nonterminal in the rule `termph ::= termph and termph` whereas the second tree could be produced by a parser which expands the right-most nonterminal first.

—They can be *deterministic* or *nondeterministic*. A deterministic parser always makes its choice of next move depending on information that it has about the grammar, the status of the tree constructed so far, and knowledge of the remaining terminals to be absorbed into the tree. It never needs to undo a move. Alternatively, a nondeterministic parser may choose a move and possibly add some structure to the tree which might subsequently have to be undone by backtracking if it leads to a situation where the tree cannot be completed.

—Deterministic parsers can differ in the amount of *lookahead* required as measured by the number of terminals symbols that must be examined before the next move can be determined.

—Depending on the combination of properties just presented, parsers can differ in the type of grammars for whose sentences they can be guaranteed to produce a syntax tree. For example, a simplistic implementation of top-down depth-first left-directional left-most expanding parsers cannot parse all sentences with respect to a grammar containing left-recursive production rules such as `t ::= t and t` as the parser would continue to expand the left-most `t` indefinitely (unless there is a mechanism to detect such looping and curtail it as discussed in Section 5.4). *General* parsers can accommodate any context-free grammar.

—Depending on the combination of properties, parsers can differ in their ability to generate all parses of ambiguous sentences.

—Depending on the combination of properties, parsers can have differing time and space complexities.

The programming language community has primarily been interested in linear deterministic parsers for analysis of unambiguous programming and command languages. Such parsers include the family LL(k), of top-down left-directional left-most expanding parsers with k-terminal lookahead, the family of deterministic bottom-up operator-precedence parsers, and the family of LR and LALR deterministic bottom-up left-directional top-down-constrained right-most reducing parsers.

On the other hand, the natural language processing community has primarily been interested in general nondeterministic parsers including the family of nondeterministic CYK bottom-up nondirectional parsers, the family of nondeterministic early-type top-down constrained dynamic programming parsers, the family of nondeterministic bottom-up Kilbury-like chart parsers, and Tomita's nondeterministic generalized GLR bottom-up breadth-first parser, which creates an efficient representation of multiple syntax trees in graph form [Tomita 1985].

A highly readable and comprehensive description of parsing techniques, which includes all of those just mentioned, is in the book by Grune and Jacobs [1990], an expanded version of which is expected to become available by the end of 2006. A theoretical treatment of parsing is given in the book by Hopcroft et al. [2000].

5.2. Use of LFP in the Implementation of Conventional Parsers for NL Analysis

Leermakers [1993] has provided an integrated treatment of deterministic and general parsing techniques in a purely functional framework. In Leermakers' approach, there is no notion of parse stack or parse matrix (the updateable data structures which are used to store control information in deterministic and general parsing techniques, respectively). The resulting purely functional treatment enables a unified view of the techniques used by the programming language and natural language communities. Leermakers shows how general recursive-ascent LR parsers can be implemented in a purely functional way and claims that there are few reasons why anyone should use anything other than recursive parsing algorithms. Although Leermaker's approach would appear to have application to the construction of natural language parsers, no one has yet made use of it.

Ljunglof [2002a, 2004] provides a comprehensive analysis of the implementation of a wide variety of deterministic and nondeterministic parsing algorithms in LFP. The algorithms include CYK parsers, Kilbury chart parsers, and LR and generalized LR parsers (leading to an approximation to Tomita's parser [Tomita 1985]). Ljunglof also provides an extensive treatment of functional parser combinators (such combinators are discussed later in Section 5.3). Ljunglof claims that lazy evaluation enables elegant and readable coding of parsing algorithms and also provides additional efficiency in that evaluation of semantic values is delayed until a complete parse has been identified.

Medlock [2002] and Callaghan [2005] have developed a GLR extension to the Haskell Happy parser generator [Marlow 2005], based on Tomita's algorithm. The extension can parse ambiguous grammars and produces a directed acyclic graph representing all possible parses. The extension began as Medlock's undergraduate project [Medlock 2002], which made use of a number of Ljunglof's ideas, and was subsequently significantly improved by Callaghan. The extension implements a GLR algorithm which can accommodate hidden as well as explicit left recursion. The Happy parser and the GLR extension both allow monadic state to be threaded through the parser, thereby accommodating languages with context dependencies. Some tests were carried out by Medlock using his version of the GLR extension on the English grammar used in the

LOLITA system (see Section 7.2 for a discussion of LOLITA). One conclusion was that efficiency would have to be improved for large parse tables. Callaghan's [2005] improved extension includes semantics and has been analyzed with respect to potential application in gene-sequence analysis, rhythmic structure in poetry, compilers, robust parsing of ill-formed input, and natural language. Callaghan and Medlock claim that the functional GLR parser is more concise, clear, and maintainable than procedural implementations. The GLR parser has been used by other researchers and is currently available as part of the Happy parser generator tool. There does not yet appear to be any extensive investigation of its use in NLI work.

Fernandes [2004] has also developed a GLR tool called HaGLR for creating parsers in pure functional programming. In this approach, the user begins by defining a grammar using a new datatype, the grammar is then passed as an argument to a function which generates the parse table, which is then passed as argument to a GLR-parsing function. Memoization is used to implement state merging. The approach is based on Tomita's original algorithm [1985], and therefore can accommodate explicit, but not hidden left recursion. Fernandes claims that lazy evaluation avoids the creation of all possible parses if they are not required. Fernandes also claims that HaGLR is faster than other implementations for ambiguous grammars and notes that GLR parsers are more compositional than LR parsers. With LR, when two parsers for two grammars are to be integrated, the combined grammar has to be manipulated before the integrated parser can be generated. This is required in order to avoid conflicts. The process is made more difficult if semantic actions are associated with the original grammars. GLR, on the other hand, allows the grammars to be combined directly. Fernandes states that this helps designers build language processors incrementally. Although HaGLR would appear to have value in building NLIs, such use has not yet been investigated in depth.

5.3. Parser Combinators

In the previous section, we described research in which LFP has been used in a conventional way to implement a range of parsers that have already been implemented in other programming languages. In this section, we describe an approach to parser construction which is unique to functional programming. The approach involves the definition and use of *parser combinators*. We describe this approach in detail as it has been used by a number of researchers to build natural language processors.

The use of parser combinators was first proposed by Burge in 1975, although he did not use the term parser combinator. The idea is to construct more complex language processors from simpler processors by combining them using higher-order functions (the parser combinators). This approach was developed further by Wadler [1985] who suggested that recognizers and parsers should return a list of results. Multiple entries in the list are returned for ambiguous input, and an empty list of successes denotes failure to recognize the input. Fairburn [1986] promoted combinator parsing by using it as an example of a programming style in which form follows function: a language processor that is constructed using parser combinators has a structure which is very similar to the grammar defining the language to be processed (as illustrated in the example given later in this section).

The simplest implementations of the use of combinators results in a top-down depth-first recursive-descent parser, with or without backtracking. We begin by describing the approach with respect to the construction of language recognizers. A is defined as a function from a sequence of input tokens to a list of sequences of output tokens. For these examples, we assume that a token is a character string.

```
type Token  = [char]            type Recognizer = [Token]->[[Token]]
```

If a parser for a sequence of tokens `t` succeeds in recognizing `t` at the front of the input sequences `(t ++ r)`, it returns the remaining tokens `r` as an element of the output list. If the recognizer fails to recognize `t` at the front of the input sequence, the output list is empty. If the input can be recognized in more than one way, the output list will contain multiple results. The following are two examples of application of a recognizer `rec_every`, which has been defined to recognize the single token "every" at the beginning of the input sequence.

```
rec_every ["every", "moon", "spins"] => [["moon","spins"]]
rec_every ["a","moon"]               => []
```

Three combinators are used to build recognizers.

(1) `term` is used to construct basic recognizers. The following is an example definition and use.

```
Recog = Recognizer
term :: token -> Recog
term w    []              = []
term w (t:ts) | w ==t     = [ts]
              | otherwise = []

rec_every   = term "every"          rec_spins   = term "spins"
```

(2) `orelse` is used as an infix operator to build alternate recognizers.

```
orelse :: Recog -> Recog -> Recog
(p `orelse` q) inp = (p inp) ++ (q inp)

rec_pnoun = (term "Phobos") `orelse` (term "Deimos") `orelse` ...
```

(3) `then1` is used to create a recognizer from two recognizers used in sequence.

```
then1:: Recog -> Recog -> Recog
(p `then1` q) inp = apply_to_all q (p inp)
         where apply_to_all q  []  = []
               apply_to_all q (r:rs) = (q r) ++ (apply_to_all q rs)

rec_pnoun_spins = rec_pnoun `then1` rec_spins
```

(4) The empty recognizer, which always succeeds, is defined as `empty x = [x]`.

These combinators can now be used to define recognizers for simple subsets of natural language. For example, consider the following recognizer for a tiny language which includes the sentences "Phobos spins", "Phobos and every moon spin", "Mars and Phobos and Deimos spin", etc.

```
rec_sentence   = rec_termphrase `then1` rec_verbphrase
rec_termphrase = rec_simpletermphrase
                 `orelse`
                 (rec_simpletermphrase
                      `then1` rec_join `then1` rec_termphrase)
rec_join       = (term "and") `orelse` (term "or")
rec_simpletermphrase = rec_pnoun `orelse` rec_detphrase
rec_pnoun      = (term "Phobos")`orelse`(term "Deimos")`orelse`(term "Mars")
rec_detphrase  = rec_det `then1` rec_noun
rec_det        = (term "every") `orelse` (term "a")
```

```
rec_noun       = (term "moon")  `orelse` (term "planet")
rec_verbphrase = (term "spin")  `orelse` (term "spins")
```

Note that precedences for the combinators could have been set to avoid the use of brackets. Also, the combinator definitions given above are highly inefficient and were chosen for clarity. More efficient implementations, which, for example, remove duplicate results, and/or use memoization to avoid repeating work, can be found in the references given later. The following are example applications of these recognizers.

```
rec_sentence ["every","moon","and","every","planet","spins","."] => [["."]]
rec_sentence ["every", "spins"] => []
rec_termphrase ["Mars","and","every","moon","spin"]
                        => [["and","every","moon","spin"], ["spin"]]
```

The first succeeds, the second fails, and the third succeeds with two results, corresponding to the two ways in which subsequences at the front of the input can be recognized as a termphrase: `["Mars"]` and `["Mars","and","every","moon"]`.

A number of advantages result from building language processors in this way.

—The combinators can be easily extended to generate parse trees and to accommodate the definition of semantic values and semantic-evaluation functions. Lazy evaluation allows semantic functions to be closely associated with the executable syntax rules without incurring huge computational overhead. This is a result of the fact that lazy evaluation only requires the semantic functions to be applied when a successful parse has been identified, not during the search process. Examples of such integration are described later in this article.

—Programs that are built using parser combinators have structures that are closely related to the structure of the grammars defining the languages to be processed. This facilitates investigation of grammars and semantic theories of language.

—The use of top-down backtracking search, which is implemented by the combinators presented earlier, leads to highly-modular parsers. Consequently, component parsers can be tested independently as illustrated by the application of `rec_termphrase`. This facilitates experimentation and reuse of components.

—The equational nature of the definitions facilitates theoretical analysis.

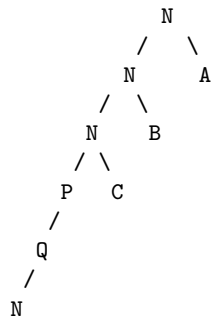
5.4. Improving Complexity and Accommodating Left Recursion

The combinator parsers described have two shortcomings for use in NLIs 1) they have exponential behavior for ambiguous grammars, and 2) they cannot be used to directly implement parsers corresponding to grammars containing left-recursive productions.

Frost and Szydlowski [1995] and Szydlowski [1996] have shown how complexity can be improved through memoization using a technique similar to that proposed by Norvig [1991] for building efficient top-down parsers in LISP. However, instead of having a global memo table, the table is threaded through function applications thereby maintaining pure functionality and modularity. Rather than making such threading explicit with consequent opportunity for error, the threading is hidden in a state monad as described in Frost [2003]. The monad encapsulates all aspects of the computation concerning the memoization process with minimal effect on the code which defines the language processors.

The problem with left recursion has been solved by Frost and Hafiz [2006] who have shown how combinator parsing can be modified to accommodate ambiguous left-recursive grammars while maintaining polynomial time complexity. The solution involves adding another table to the state which is threaded through all calls of all

functions implementing the component parsers. The new table counts the number of times each parser is applied to the same input. For nonleft-recursive parsers, this count will be at most one as the memotable lookup will prevent such parsers from ever being applied to the same input twice. However, for left-recursive parsers, the count is increased on recursive descent (owing to the fact that the memotable is only updated on the recursive ascent). Application of a parser `N` to an input `inp` is failed whenever the application count exceeds the length of the remaining input plus 1. When this happens, no parse is possible (other than spurious parses which could occur with circular grammars). As illustration, consider the following branch created during the parse of two remaining tokens on the input:



where `N, P, Q` are nonterminals, `A, B, C` are sequences of terminals and nonterminals, and the left-recursive grammar is of the form

```
N := N A | N B | P C | ..      P := Q .. | ..          Q := N .. | ..
```

The last call of `N` should be failed owing to the fact that, irrespective of what `A`, `B`, and `C` are, either they must require at least one input token, or else they must rewrite to `empty`. If they all require a token, then the parse cannot succeed. If any of them rewrite to `empty`, then the grammar is circular (`N` is being rewritten to `N`), and the last call should be failed.

Notice that simply failing a parse when a branch is longer than the length of the remaining input is incorrect as this can occur in a correct parse if recognizers are rewritten into other recognizers which do not have token requirements to the right. For example, the parse should not be failed at `P` or `Q` as these could rewrite to `empty` without indicating circularity.

A major advantage results from this approach, namely, the memotable is a compact polynomial representation of the potentially exponential number of parse trees. This compact representation is similar to the graphical data structure generated by Tomita's algorithm [1985]. However, as discussed in Frost and Hafiz [2006], an additional advantage of using LFP is that lazy evaluation allows the memotable to be built only as needed for the question at hand. For example, if the question concerns just the location of term phrases in the input, then the parser only generates begin and end points in the memotable. If the first parse tree is the just one required to be made explicit, then the memotable will contain just those subtrees that are needed.

The approach developed by Frost and Hafiz [2006] was influenced by the methods proposed by Kuno [1965], Shiel [1976], Lickman [1995], and Johnson [1995] for dealing with left recursion in top-down parsing.

5.5. A Short History of Parser Combinators and Their Use in NLIs

Parser combinators have a long history.

Burge [1975] was the first to suggest the use of higher-order functions to create complex language processors from simpler components.

The use of lists to accommodate failure or multiple successful solutions in search problems appears to have been first proposed by Turner [1981]. Wadler [1985] was the first to apply this method to parsing and introduced the notion of failure as an empty list of successes which is central to the definition of parser combinators.

Fairburn [1986] used parser combinators as an example in advocating a programming style in which form follows function.

Frost [1992] defined combinators which enable the construction of language processors as executable specifications of attribute grammars.

Hutton [1992] provided a comprehensive description of parser combinators and demonstrated their use in the construction of a parser for program code.

Fokker [1995] demonstrated how parser combinators can be used to analyze arithmetic expressions.

Hill [1996] defined parser combinators for expressions with precedences and associativities.

Panitz [1996] provided a proof of termination for combinator parsers using as an example the combinators of Frost and Launchbury [1989].

Frost and Szydlowski [1995] and Szydlowski [1996] demonstrated how memoization can be used to reduce the complexity of combinator parsers for ambiguous grammars.

Patridge and Wright [1996] defined combinators which can be used to build efficient predictive parsers which return values that are either parse trees or an indication of the cause of a parsing error.

Swierstra and Duponcheel [1996] defined combinators which produce error-correcting parsers for LL(1) grammars. However, the approach is incompatible with the use of a monadic interface. Hughes [2000] suggests a potential solution to this deficiency by introducing arrows as a generalization of monads.

Hutton and Meijer [1998] wrote a tutorial on a monadic approach to the definition of parser combinators and discussed the advantages which result from this approach.

Koopman and Plasmeijer [1999] showed how the efficiency of combinator parsers can be substantially increased by use of continuations to avoid the creation of intermediate data structures and the introduction of an exclusive `orelse` combinator to be used to limit backtracking where it is known that only one alternative can succeed.

Leijen and Meijer [2001] developed Parsec, a library of monadic parser combinators built in Haskell. Parsec can be used to build industrial-strength language processors, including compilers which have appropriate efficiency and error handling. Leijen and Meijer also state that an advantage of monadic parser combinators is that they can parse context-sensitive grammars, whereas the earlier parser combinators are restricted to context-free grammars.

Ford [2002] developed the Packrat parser which parses LL(k) and LR(k) grammars in linear time.

Frost [2003] demonstrated how the somewhat-error-prone method of memoizing parser combinators developed by Frost and Szydlowski can be systematized through a process of monadic memoization.

Frost and Hafiz [2006] used monadic memoization to accommodate left recursion in top-down parser combinators (see Section 5.4).

Parser combinators have received significant attention from the functional programming community and have been used extensively in programming-language prototyping. Their use in natural language processing has been more limited. The following is a summary of the use of parser combinators in NLI. More information is given later in the article.

Frost and Launchbury [1989] defined parser combinators to implement a natural language database-query processor based on a set-theoretic version of Montague semantics. The combinators were subsequently extended to allow language processors to be constructed as executable specifications of attribute grammars (that work is described in more detail in Section 7.2).

A significantly extended version of the combinators of Frost and Launchbury [1989] was used in an early implementation of the LOLITA natural language processing system [Garigliano et al. 1992] which is described in more detail in Section 7.1.

Lapalme and Lavier [1990, 1993] defined parser combinators to build a workbench for investigating Montague-like theories of language (described in Section 7.3).

Ljunglof [2002b] published a brief argument supporting the use of LFP in natural language processing. As an example, he developed what he refers to as a Montague-style parser for a mini NL language using a set of parser combinators. The processor takes NL phrases and returns expressions of first-order predicate calculus. For example, `parse sentence (words "sas serves every city in Europe")` returns: `forall x (city,x) & in(x,Europe) => serves(sas,x).`

Van Eijck [2003] defined new parser combinators that can be used to build parsers which can accommodate phrases containing dislocation phenomena such as *left extraction* in natural language. Left extraction occurs when a component of a phrase is missing and some other component to the left of it contains the missing part. For example, relative clauses contain left extraction, for example, "I knew the man that the woman sold the house to". One of van Eijck's combinators, `expectDPgap`, is such that, for example, when it is applied to the parser for sentences, will return a parser for relative clauses. The approach can also create parsers for queries such as "What did they break it with?" and "With what did they break it?". Van Eijck [2004] has also implemented a "deductive" Early-like general parser in Haskell but provides no discussion of its use in natural language processing.

Pace [2004] defined parser combinators to accommodate use of context in natural language parsing. The combinators are implemented in Haskell and make use of the built-in monad support. Pace uses his combinators to build parsers for Maltese, where, for example, the rules for constructing a term phrase from a determiner (e.g., "the") and a noun are more complicated than in English, involving morphological rules to aid pronunciation of the words with grammatical rules which do not have a straightforward compositionality. The context is represented in a state monad which is threaded through the component parsers. Pace appears to be the first to use monadic combinators to implement context-sensitive parsers for natural language.

5.6. Use of LFP in Grammar Analysis

Jeuring and Swierstra [1994] have formally specified a number of bottom-up grammar analysis problems, and then systematically derived LFP programs for bottom-up grammar analysis from the specifications. One example problem is to determine if a given nonterminal derives the `empty` string.

5.7. LFP and the Construction of Morphologies

A *morphology* is a system which explains the internal structure of words. Regular nouns have relatively simple morphological structure. For example, "cat", "cat"++"s", and "cat"++""++"s", whereas irregular nouns and verbs have more complex morphology. The morphology of a particular language can be defined using a set of inflection tables. For example, for the Latin word "*rosa*", meaning rose,

	Singular	Plural		Singular	Plural
Nominative	rosa	rosae	Vocative	rosa	rosae
Accusative	rosam	rosas	Genitive	rosae	rosarum
Dative	rosae	rosis	Ablative	rosa	rosis

Writing out a table for every word in a language would result in hundreds of thousands of entries. Consequently, more efficient representations are required. One approach is to create tables for some words, which are then used as paradigms for the definition of the morphology of other words.

The conventional approach to morphological recognition is to compile the tables into finite state automata, and then to parse words as regular expressions. As an alternative, Pembeci [1995] has built a morphological analyzer for Turkish using parser combinators implemented in Miranda and claims that the analyzer can parse approximately 99% of all word forms in Turkish. Pembeci also claims that all morphological processes have been implemented, and that a number of advantages result from use of parser combinators, including clarity and modifiability of the code.

Forsberg [2004], and Forsberg and Ranta [2004] have developed an alternative approach, called Functional Morphology which is implemented in Haskell. It is based on Huet's toolkit, Zen, [Huet 2003, 2004] which Huet used to build a morphology for Sanskrit. In this approach, inflection parameters are defined using algebraic data types, and paradigm tables are implemented as finite functions defined over these types. As illustration, consider the following example given in Forsberg and Ranta [2004].

```
data Number = Sing | Plural
data Case   = Nominative|Vocative|Accusative|Genitive|Dative|Ablative
data NounForm = NounForm Number Case
type Noun     = NounForm -> String

rosa :: [(NounForm, String)]
rosa =
[(NounForm Sing Nominative, "rosa"), (NounForm Sing Vocative,   "rosa"),
 (NounForm Sing Accusative, "rosam"),(NounForm Sing Genitive,   "rosae"),
 (NounForm Sing Dative,     "rosae"),(NounForm Sing Ablative,   "rosa"),
 (NounForm Plural Nominative, "rosae") ...

rosaParadigm :: String  -> Noun
rosaParadigm rosa (NounForm n c) =
 let rosae = rosa ++ "e"
     rosis = init rosa ++ "is"
     in case n of Singular ->  case c of  Accusative ->  rosa ++ "m"
                                          Genitive   ->  rosae
                                          Dative     ->  rosae
                                          _          ->  rosa
                  Plural   ->  case c of  Nominative ->  rosae
                                          Vocative   ->  rosae
                                          Accusative ->  rosa ++ "s"
                                          Genitive   ->  rosa ++ "rum"
                                          _          ->  rosis
```

One advantage of using functions is that the morphology of other words can now be defined succinctly in terms of the paradigms. For example,

```
dea :: Noun
dea nf = case nf of NounForm Plural Dative    ->  dea
```

```
                NounForm Plural Ablative  ->  dea
                _                         ->  rosaParadigm dea nf
        where dea = "dea"
```

Lists are used to accommodate free variation in which two or more words that have the same meaning are spelled differently, for instance, "domus" and "domos" (home) as well as missing forms corresponding to tables which have missing values. Haskell's string-handling capabilities are used to accommodate features that are difficult to define using regular expressions. For example, dropping one of the letters when the last letter of a word and the first letter of an ending coincide.

Functional Morphology has been used to define morphologies for a number of languages including Italian [Ranta 2001], Spanish [Andersson and Soderberg 2003], Russian [Bogavac 2004], Swedish and Latin [Forsberg and Ranta 2004].

6. USE OF LFP IN SEMANTIC ANALYSIS

6.1. Implementing Montague-Like Semantics in LFP

We begin, in this section, by illustrating the ease with which computationally-tractable versions of Montague-like semantic theories can be implemented in LFP by the direct encoding of the higher-order functional semantics. This is analogous to the use of Prolog to encode first-order semantic theories.

The approach borrows many insights from Montague, but differs in that common nouns and phrases which denote characteristic functions of sets in Montague denote the sets themselves, and all other denotations are modified accordingly. This is necessary in order to efficiently compute answers when queries are evaluated with respect to a database. For example, in a direct implementation of Montague semantics, evaluation of the query "Does every moon spin?" would involve application of the characteristic functions denoted by "moon" and "spin" to each entity in the universe of discourse. In the set-theoretic version, "moon" and "spin" denote the sets of entities directly, and the query is evaluated by determining if the first set is a subset of the second.

The following illustrates how the set-theoretic semantics can be used as the basis for a small query processor implemented in Haskell. The implementation begins by introducing the internal representations of entities through a user-defined type as follows, where the code `deriving (Eq, Show)` causes `Entity` to inherit properties of `Eq`, enabling use of == for testing equality, and of `Show` for printing.

```
data Entity = Earth | Mars| Phobos | Deimos |...     deriving (Eq, Show)
```

Next, the denotations of common nouns and intransitive verbs are represented as lists of entities. For example,

```
spins, planet, moon, person :: [Entity]
spins  = [Mars, Earth, Phobos, ..      planet = [Mars, Earth, Mercury, ..
moon   = [Luna, Phobos, Deimos, ..     person = [Hall, Kuiper, ..
```

Next, the denotations of proper nouns are represented as functions from entity sets to booleans, using the built-in function `elem` which tests for membership in a list. Note that lower case is used for identifiers representing the denotation of words and an initial upper-case letter for identifiers of the internal representation of entities. For example, "Mars" is the word in the concrete syntax, `mars` is its semantic value (a function), and `Mars` is the internal representation of the entity associated with the word "Mars" (see the following). Denotations of quantifiers are represented as higher-order functions. For example, assuming that `subset` and `intersect` have been defined appropriately,

```
mars, phobos :: [Entity] -> Bool
mars s  = Mars ‘elem‘ s          phobos s = Phobos ‘elem‘ s  ...

every, a :: [Entity]->[Entity]->Bool
every s t = s ‘subset‘ t         a  s t = s ‘intersect‘  t /= []
```

These definitions can be used directly in composite semantic expressions, which can be entered at the command line. For example, `mars spins => True` and `every planet spins => False`

A more complex definition is required for the denotation of the word “and”.

```
(f ‘and‘ g) = h where h s = (f s) && (g s)
```

The value of an expression `f ‘and‘ g` is a function `h` which takes a list of entities `s` as input and which returns the Boolean value of the result returned by forming the logical conjunction `&&` of the values of `(f s)` and `(g s)`. Hence, `(mars‘and‘(every moon)) spins => True` The denotation of the word “or” can be defined similarly using disjunction in place of conjunction, and the word “that” has the denotation `that = intersect`. Conversion of the denotation of the transitive verb “discovered” given in Section 3.3 to a set-theoretic version yields the denotation `discovered` defined as follows:

```
discovered p = [x | (x, image_x) <- collect discover_rel, p image_x ]
    where discover_rel=[(Hall, Phobos),(Hall, Deimos),(Kuiper, Nereid)...
```

The `collect` function is defined such that it returns a new binary relation containing one binary tuple `(x, image_x)` for each member of the projection of the left-hand column of `discover_rel`, where `image_x` is the image of `x` under the relation `discover_rel`. Example applications of `collect` and `discovered` are:

```
collect discover_rel => [(Hall,[Phobos,Deimos]),(Kuiper,[Nereid...]),...
discovered phobos
     => [x |(x,image_x) <- [(Hall,  [Phobos, Deimos]),
                            (Kuiper,[Nereid...     ])
                            ...],                      phobos image_x]
     => [Hall]
```

Passive forms of verbs such as `was_discovered_by` can be accommodated by defining them as just presented except that the order of the values in the tuples in the associated binary relation is reversed.

The resulting minisemantics is highly compositional in the sense that the only rule of composition is function application, and the order of application is determined by the syntactic structure of the query. Example query and subquery evaluations are:

```
(Hall ‘and‘ Kuiper) (discovered (a moon))   => True
(moon ‘that‘ (was_discovered_by
                     (Hall ‘or‘ Kuiper)))
                                           => [Phobos,Deimos,Nereid...
((a moon) ‘and‘ (every planet))
             (was_discovered_by (a person)) => False
every moon                                 => <function>
```

Note that the denotation of the phrase “every moon” is of the same type as the denotation of the proper noun “Mars”, given earlier. This is consistent with Montague’s approach which dictates that words and phrases of the same syntactic category should denote semantic values of the same type. Note also that denotations of words can be defined in terms of the meaning of other words and phrases. For example,

```
discoverer = person `that` (discovered ((a moon) `or` (a planet)))
```

One of the problems with the semantics described so far is that it cannot accommodate sentences such as "Phobos orbits and Deimos orbits Mars". The reason is that the phrases "Phobos orbits" and "Deimos orbits" cannot be given straightforward denotations using function application because the type of the denotations of "Phobos" and "Deimos" is `[Entity] -> Bool`, which cannot be applied to the denotations of "*orbits*" which is of type `([Entity] -> Bool) -> [Entity]`. The rules of Combinatory Categorial Grammar described in Section 3.5 suggests the following solution to this problem: 1) introduce a new syntactic category `termphrase_transverb ::= termphrase transverb`, 2) construct the denotation of a `termphrase_transverb` by composing the denotations of the two components on the right of the rule: `denotation_of_termphrase . denotation_of_transverb` As an example, consider the sentence "Phobos orbits and Deimos orbits Mars". The additional grammar rule, together with other rules which refer to the new category, would cause the sentence to be parsed as follows: `((Phobos orbits) and (Deimos orbits)) Mars`, and the semantic rule would result in the following interpretation.

```
((phobos . orbits) `and` (deimos . orbits)) mars
 => ((phobos . orbits) mars) ((deimos . orbits) mars)
 => (phobos (orbits mars)) (deimos (orbits mars))
 =>                        True
```

This is not an entirely satisfactory solution as we need function composition in addition to prefix and infix function application. However, the approach accommodates a wide range of sentences such as "Hall discovered and Mars is orbited by Phobos", "Phobos orbits and Deimos orbits and Miranda orbits a planet", etc.

The semantics that has been presented in this section is linguistically simple, yet it serves to illustrate the ease with which efficient versions of Montague-like theories can be represented in LFP. An implementation of this semantics was first presented by Frost and Launchbury [1989] who integrated it with parser combinators, written in Miranda, in order to create an efficient natural language database-query processor. Independently, and around the same time, Lapalme and Lavier [1990, 1993] implemented a subset of Montague grammar in Miranda using parser combinators in order to create a framework for experimentation. These two projects are described in more detail in Sections 7.2 and 7.3, respectively. In both cases, the researchers claimed that higher-order functions and lazy evaluation facilitated the creation of highly modular systems.

6.2. Use of LFP to Investigate and Extend Semantic Theories of Natural Language

In addition to implementing theories, researchers have also used LFP to investigate extensions to those theories for use in NLIs.

6.2.1. Efficient Accommodation of Arbitrary Negation. Despite comprehensive analysis of negation by linguists, for example, [Iwanska 1992], the creation of a computationally-tractable compositional method for accommodating arbitrary negation in NLIs has proven to be difficult. The problem can be illustrated by considering the following queries with respect to a relational database containing data about which moons orbit which planets: "Does Luna not orbit Mars?" and "Does Sol not orbit Mars?" Montague-like compositional semantic theories, such as those described in Section 6.1, will return the correct answer for the first query but the wrong answer for the second query (with respect to the closed-world assumption, which is appropriate for many applications). This is because the orbits relation does not contain `Sol` in its left-hand column due to

the fact that Sol does not orbit anything. Therefore, `Sol` is not returned in the list of entities which is the interpretation of "not orbit Mars". One solution to this problem is to extend the orbit relation to include `(x, "nothing")` for all entities `x` in the domain of discourse which do not occur in the left-hand column. This is clearly impractical for all but very small databases and is useless for databases with infinite domains. Frost and Boulos [2002] have developed a solution to this problem and have implemented an example in LFP. The basic idea is that potentially huge or infinite sets denoted by constructs involving negation are represented using set-complement notation in which the type `CSET` is defined as follows: `data CSET = SET [ENTITY] | COMP [ENTITY].`

The basic set operators are redefined, as exemplified in the following, and the denotations of words are redefined accordingly. For example,

```
c_intersect (SET  s) (SET  t)    = SET  (s `intersect` t)
c_intersect (SET  s) (COMP t)    = SET  (s -- t)
c_intersect (COMP s) (SET  t)    = SET  (t -- s)
c_intersect (COMP s) (COMP t)    = COMP (s `unite`  t)

no s t = s `c_intersect` t == []

non (SET s)  = COMP s
non (COMP s) = SET s
```

The approach accommodates arbitrarily nested quantification and negation.

```
every (thing `that` (orbits (no moon))) (orbits (no planet))       => False
a (non moon) (orbits Sol)                                          => True
every moon (orbits (no moon))                                      => True
Sol (orbits (a (non moon)))                                        => False
not (every moon) (is_orbited_by Phobos)                            => True
a (moon `that`(was_discovered_by Hall)) (does (not (orbit Earth))) => True
moon `that` (was_discovered_by Hall) => SET  [Phobos, Deimos]
orbits (no planet)                   => COMP [Phobos,Deimos, Nereid...]
```

6.2.2. Accommodating Transitive Verbs of Arity Greater Than 2. The semantic theory described in Section 6.1 can only accommodate transitive verbs of arity two. It cannot handle queries such as "When, and with what, did Hall discover Phobos?". Montague gives little help in this respect. Roy [2005] and Roy and Frost [2005] have developed an approach which goes some way towards solving this problem for queries that are interpreted with respect to first-order nonmodal nonintensional databases. The basic idea is that atomic semantic values (entities, etc.) are represented as attributes of the same type by applying user-defined value constructors to them (such constructors include `SUBJ, OBJ, IMPLEMENT, TIME` etc.) Relations are represented as lists of lists of attributes. Consequently, all relations are of the same type irrespective of arity. All denotations are modified accordingly such that phrases denote lists of attributes rather than simply truth values or lists of entities, etc. For example, interpretation of the phrase "Hall discovered a moon" returns the list

```
[[SUBJ Hall,OBJ Phobos,IMPL Telescope, TIME ...],[SUBJ Hall,OBJ Deimos...
```

Phrases such as "when did" and "with what did" are then used as filters to return answers to specific questions.

6.2.3. A Uniform Treatment of Adjectives. Despite their simple syntactic form, adjective-noun combinations seem to have no straightforward semantic method that parallels

the simplicity of their syntax. For example, consider the phrases "beautiful dancer", "fake gun", "tall jockey", and "former senator". One view is that adjectives belong to a semantically motivated hierarchy. This has the consequence that a uniform treatment of adjectives is difficult. In contrast to this view, Abdullah and Frost [Abdullah 2003; Abdullah and Frost 2005] have developed a uniform semantics based on typed sets. Entities belong to a set only when associated with a type. For example, the set `beautiful = {Mary:woman, Jane:dancer}` states that Mary is beautiful as a woman and Jane dances beautifully. The semantics makes use of typed-set operators to ensure that only valid deductions can be made. For example, even if Mary is also in the set of dancers, it would not follow that she "dances beautifully". Phrases that involve privative adjectives, such as "fake gun" are dealt with by treating fake guns as belonging to the set of guns but lacking some intrinsic properties so that the denotation of such phrases is obtained by a modified form of intersection. In this approach, regular adjectives such as "red", "angry", or "skillful" and privative adjectives such as "fake" or "former" have one thing in common: they both constrain the domain denoted by the noun that follows it. They differ in the means of doing it, regular adjectives highlight some properties of the noun, while privative adjectives mask some properties. This approach was developed through experimentation in Miranda and a small database-query processor has been implemented to demonstrate its viability.

6.2.4. Dealing with Dynamic Contexts. Various compositional theories have been developed to model dynamic context in interpreting natural language constructs, such as "Some student studied hard for some subject; he did well in it it.", "Few students go on to doctoral studies; they are highly motivated", and "Few students completed the assignment; they are very lazy", which involve pronominal reference, anaphoric linking, and dynamic scoping. The theories attempt to develop an interpretation of the first phrase, which is then merged with the interpretation of the second phrase. Most of these theories replace the static variable-binding scheme of predicate logic with a dynamic binding such that interpretations involve relations between variable states in the model. Van Eijck [2001] has developed an alternative theory, called incremental dynamics (ID), which differs from other theories in that it uses a variable-free representation of quantifiers. The approach appears to have been motivated by combinatory logic, a variable-free representation of lambda calculus which is used as a theoretical basis for the implementation of some LFP languages. The basic idea behind ID is that variables are replaced by indices into contexts where a context can be thought of as a data structure representing information on entities, etc., gathered from some subcomponent of the phrase. Existential quantifiers push entities into contexts and pronouns select entities from contexts. Anaphora resolution involves using syntactic clues to determine where to search in a set of contexts. Van Eijck and Nouwen [2002] have developed an example implementation of an ID-based natural language interpreter in Haskell, making particular use of polymorphic types to represent contexts.

6.3. Use of Types to Analyze Natural Language

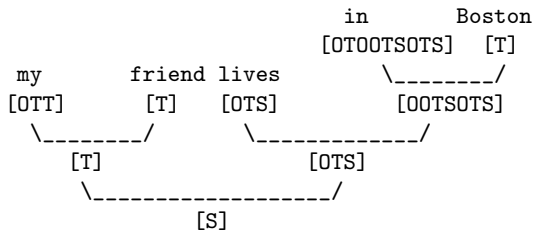
Universal Applicative Grammar (AUG) is a linguistic theory which was developed by Shaumyan [1987]. The structures and rules which define a particular language such as English, are called a phenotype. The universal structures and laws which underlie all natural languages are collectively called the genotype. From a computer science perspective, the genotype is in some ways analogous to abstract syntax and the phenotype to concrete syntax. The genotype is defined in terms of predicates, terms, predicate modifiers, term modifiers, and rules of combination which are based on combinatory

logic. These rules are constrained by a set of semiotic principles which are intended to explain universal linguistic features.

For example, the passive form of the sentence "Hall discovered Phobos" is created by application of a rule in the phenotype of English which reverses the order of Hall and Phobos and adds "was" and "by", giving "Phobos was discovered by Hall". This rule of passivization does not hold in languages such as Russian or Latin where passivization is achieved by use of case endings. In AUG, a universal law of passivization is stated in terms of operations on the structures of the genotype.

In AUG, phenotype and genotype grammars are defined in terms of types and operator/operand relations, corresponding to categories and function/argument relations in Categorial Grammar, using a notation analogous to that of Categorial Grammar in which a phrase of type `Oxy` combines with a phrase of type `x` to generate a phrase of type `y`.

In Section 5, we described systems in which grammars are used to direct the syntactic analysis of the natural language input. However, in some applications, it may not be feasible to define a grammar that covers all of the ways in which users might phrase their input, and, in some applications, the system may be required to be sufficiently robust to be able to process input that would usually be regarded as being grammatically incorrect. One solution to this problem which is based on AUG is to use types to parse natural language [Jones et al. 1995; Shaumyan and Hudak 1997]. For example, consider the phrase "my friend lives in Boston". The word "friend" might be of type `T` (for term) and "my" might be of type `OTT` meaning that it takes a phrase of type `T` and returns a phrase of type `T`. The assignment of a particular order of function application to a phrase can be thought of as a parse of that sentence. For example,



Parsing may now be thought of as identifying all well-typed tree structures for adjacent phrases in the input. The advantage of this approach is that it does not require a grammar, and it can accommodate queries with less constrained word order. A disadvantage is the exponential size of the search space. However, Jones et al. [1995] have shown how memoization can be used to improve the efficiency of the process. The approach has been implemented in Haskell and is being used to investigate the inference of types for words not already in the dictionary, use of punctuation, and other aspects of NL processing. More comprehensive descriptions of AUG can be found in Shaumyan [1987], Sypniewski [1999], and Shaumyan and Segond [1994], which include comparisons of AUG with Combinatory Categorial Grammar.

6.4. Use of LFP to Model Semantic Ontologies

In addition to the use of LFP to represent compositional theories of language, it has also been proposed as a means for modeling ontologies. An ontology is a collection of definitions of related semantic concepts. Ontologies are necessary to support the analysis and reasoning that is required in, for example, advanced natural language information retrieval systems. WordNet, which is a lexical reference system developed at The Cognitive Science Laboratory at Princeton University (`http://wordnet.princeton.edu/`)

is a widely-used ontology. In WordNet, English words are grouped into synonym sets representing underlying lexical concepts which are related through various semantic functions.

Kuhn [2002] has noted that, although ontologies that are based on hierarchies such as Wordnet can approximate human similarity judgments, they fail to represent other semantic relationships. He gives as example the relationship between boathouses and houseboats. After simplification, Kuhn derives the following subhierarchies from WordNet:

```
boathouse-(at edge of river or ...
=>barge-(a boat with a flat ...
 =>house-(a building in which ...
  =>building-(a structure that ...
   =>structure-(a thing constructed .
    =>artifact-(a man-made object)
     =>object-(a physical ...
      =>entity-(anything that exists
```

```
houseboat-(a barge that is ...
=>boat-(a small vessel for travel ..
 =>vessel-(a craft designed for ...
  =>craft-(a vehicle designed  ...
   =>vehicle-(a conveyance that ...
    =>conveyance-(something that ...
     =>instrumentality-(an ...
      =>artifact-(a man-made ...
       =>object-(a physical ...
        =>entity-(anything ...
```

A commonly-used measure of similarity is the number of steps separating two concepts from a common node in the hierarchy. Using this, `boathouse` would be 12 steps away from the concept of `houseboat`, with `artifact` as the common node. Kuhn argues that this measure does not adequately capture the similarity/dissimilarity of these two concepts with respect to their sheltering function and their relationship with people and water. As a solution, Kuhn proposes the use of a technique called *conceptual integration* and its formalization using Haskell class declarations. The basic idea behind conceptual integration is that semantic entities belong to conceptual categories by virtue of admitting certain operations. For example, an entity is in the category `house` if it affords (Kuhn's terminology) shelter to other concepts.

In order to formalize the representation of his ontology, Kuhn uses Haskell type classes (as discussed in Section 4.4) together with Haskell multiparameter class definitions, which are supported by some implementations of Haskell, to represent semantic categories whose members share behavior. He begins by specifying examples of image schemata.

```
class Containers a b where
 insert        :: b -> a b -> a b
 remove        :: b -> a b -> a b
 whatsin       :: a b ->  [b]

class Surfaces a b where
 put           :: b -> a b -> ab
 takeoff       :: b -> a b -> a b
 whatsOn       :: a b -> [b]
```

```
class Contacts a b where
 attach        :: b -> a b -> a b
 detach        :: b -> a b -> a b
 whatsAt       :: a b -> [b]

class Paths a b c where
 move     :: c -> a b c -> a b c
 origin,destination :: a b c -> b
 whereIs      :: a b c -> c -> b
```

The operations afforded by the types in the class are described through their signatures. For example, the `insert` operation puts a thing of type `b` into a container of type `a b` and returns a container holding that thing. Query functions return lists of things contained, supported, or attached.

Next, three auxiliary concepts are defined: `People` (as house inhabitants and boat passengers), `HeavyLoads` (as defining the capacity of barges), and navigable `WaterBodies` (as transportation media) which are constrained to be subclasses of `Surfaces`. The final part of the formalization is the specification of all concepts above houseboats and

boathouses in the modified WordNet hierarchies:

```
class People p
class Surfaces w o => WaterBodies w o
class HeavyLoads l

class Containers h o                            => Houses   h o
class (Surfaces  v o, Paths a b (v o)           => Vehicles v o a b
class (Vehicles  v o a b, WaterBodies w (v o)) => Vessels  v o a b w
class (Vessels   v p a b w, People p)           => Boats    v p a b w
class (Boats     v p a b w, HeavyLoads p)       => Barges   v p a b w
```

Now, `BoatHouses` can be defined as `Houses` which are used to store `Boats` and which are located at the edge of a body of water on which the boats can move, and `HouseBoats` can be defined as barges used as houses for people.

```
class (Houses h (v p), Boats v p a b w, Contacts w (h (v p)))
                                          => BoatHouses h v p a b w

class (Barges v p a b w, Houses v p, People p)  => HouseBoats v p a b w
```

One advantage claimed for this approach is that the ontology can be checked for errors using the Haskell type-checking system. Kuhn also claims that with instantiations of types of classes and with operations defined on those classes, semantic properties can be determined. As example, he states that, in the previous ontology, it can be shown that a passenger on a boat in a boathouse cannot be said to be an inhabitant, whereas a passenger on a houseboat can. However, Kuhn does not give details of how such reasoning could be automated nor does he compare his proposed approach to existing ontologies such as WordNet.

Frank [2001] has also used Haskell to implement a model of a tiered ontology for a geographical information system but makes no comment on the advantages of using the LFP paradigm.

7. NLI SYSTEMS BUILT USING LFP

7.1. LOLITA

LOLITA is a Large-scale, Object-based, Linguistic Interactor, Translator and Analyzer that was under development from 1986 up to 1989 by Roberto Garigliano and other members of the Natural Language Engineering Group at the University of Durham. LOLITA is based on three main capabilities: 1) conversion of English text to an internal semantic network called SemNet, 2) inferences in SemNet, and 3) conversion of parts of SemNet to natural language output.

LOLITA was originally built in Miranda [Garigliano et al. 1992]. However, it is now implemented in over 50,000 lines of Haskell and 6,000 lines of C. It is one of the largest programs written in an LFP language. LOLITA was entered in DARPA's Message Understanding Conference Competitions MUC-6 [Morgan et al. 1995] and MUC-7, and participated successfully. The results of MUC-6 are reported and comprehensively analyzed in Callaghan's doctoral thesis [Callaghan 1997]. According to Callaghan, the parser for LOLITA, prior to the MUC-6 competition, was developed from the parser combinators of Frost and Launchbury [1989] with considerable extensions. Subsequently, it was replaced by a more comprehensive natural language parser written in C which generated compact graph-based representations of parse trees similar in some ways to those generated by Tomita's algorithm [Tomita 1985]. The parser translates English

text to one or more disambiguated structures which are then processed and added to the graph-based semantic network.

LOLITA's semantic net serves many purposes and is used to represent ontological hierarchies and lexical information (from WordNet), prototypical events, general knowledge, and knowledge gained from previous analysis of natural language input [Short et al. 1996]. A type-theoretic semantics for SemNet has been developed [Shiu et al. 1996; Shiu 1997]. LOLITA provides various forms of reasoning in order to make inferences from its semantic network. These inferences are used to support parsing and other natural language processing tasks. This reasoning includes 1) inheritance, in which nodes in the net can gain information from their neighbors, 2) analysis of semantic distance, which is used to determine where to place new nodes, and 3) analogy, which supports inference based on similarity of semantic structures [Long and Garigliano 1994].

A number of prototype applications have been built using LOLITA. These include an information-extraction system which summarizes text [Garigliano et al. 1993], a Chinese tutoring system [Wang 1994], a natural language generation system for English [Smith et al. 1994; Smith 1997] and for Spanish [Fernandez 1995], a metaphor processor [Heitz 1996], a discourse planner [Reed et al. 1997], a natural language database-query processor, and an information-extraction system for equity derivatives trading [Constantino 1999].

Not only has LOLITA demonstrated the viability of LFP for building large systems, it has also demonstrated the suitability of LFP for rapid prototyping, which is necessary in the continually evolving natural language research domain. Lazy evaluation was found to be essential for performance in that it allowed only the best semantic subnets to be evaluated—the input was first decoded by the parser which constructed the parse graph only as required to generate the results. Maintaining this laziness was one of the challenges faced when LOLITA was successfully parallelized by Loidl et al. [1997].

7.2. Attribute-Grammar Programming Environments

An attribute grammar (AG) is a context-free grammar augmented with associated semantic rules. Attribute grammars can be compiled into programs which parse and evaluate their input according to the grammar specification. As an alternative to compiling AGs, extended parser combinators can be defined so that language processors can be constructed as executable attribute grammars directly in the programming language. A comprehensive survey of attribute grammars and attribute grammar programming environments is given in Paakki [1995].

Johnsson [1987] was the first to provide support for attribute grammar programming in LFP. He added a new case-like structure to a lazy functional language to express attribute dependencies over data structures. The approach used lazy evaluation to hide the two-pass aspect of many tree-processing problems (e.g., scanning a parse tree to build a context, and then subsequently scanning the tree again to make use of that context). Around the same time, Udderborg [1988] built a purely functional parser generator which accepts specifications of a general class of attribute grammars as input and which returns language processors coded in LML as output. Udderborg claimed that lazy evaluation was necessary to accommodate certain types of circular attribute dependencies. A third approach was investigated by Augusteijn [1990] who developed the Elegant attribute grammar programming language. The name is an acronym for Exploiting Lazy Evaluation for the Grammar Attributes of Non-Terminals. Elegant started as a compiler generator based on attributed grammars but grew to become a complete programming language. The design of Elegant was inspired by the abstraction mechanisms found in lazy functional programming languages.

Although attribute grammars are clearly relevant to the specification and construction of natural language interfaces, Johnsson, Udderborg and Augusteijn did not discuss the potential use of their functional attribute grammar systems in such work.

Frost and Launchbury [1989] appear to have been the first to consider the use of functional attribute grammars in NLIs. They defined a set of parser combinators, similar to those described in Section 5.3 but which allow a single attribute to be associated with each production rule in the grammar. The resulting simple form of executable attribute grammar was used to implement a natural language database-query processor based on a set-theoretic version of Montague semantics, similar to that described in Section 6.1.

The combinators of Frost and Launchbury were extended to accommodate dependencies between inherited and synthesized attributes in a system called the Windsor Attribute Grammar Programming Environment (W/AGE) constructed in Miranda [Frost 2002]. The environment allows syntax and semantic rules to be defined together in a form that closely resembles attribute grammar notation.

As example of the use of W/AGE, consider the following extracts (converted to Haskell) from an 800–line Miranda program which can answer hundreds of thousands of simple queries such as "Who discovered a moon that orbits a planet that is orbited by Phobos or Nereid?". The program begins with a declaration of the types of semantic attributes that are to be computed for different syntactic categories. For example, in the following, where `Es` stands for entity set:

```
data Attribute  =   SENT_VAL Bool
                  |  NOUNCLA_VAL Es
                  | ADJ_VAL Es
                  |  TERMPHRASE_VAL (Es -> Bool) ...
```

A dictionary is then created defining the vocabulary of the input language. For each word, the entry indicates the syntactic category and its meaning. Words can also be defined in terms of other words or phrases. Basic interpreters are then defined in terms of the dictionary entries. For example,

```
dictionary = [("moon",       cat_cnoun, [NOUNCLA_VAL set_of_moons])...
              ("discoverer", cat_cnoun, meaning_of nounclause "person
                                          who discovered something")...
cnoun  = dictionary_category cat_cnoun
```

Two attribute grammar combinators `‘orelse‘` and `structure` are then used to define the syntax and associated semantic rules. For example, the following rule for simple noun clauses states that a simple noun clause is either a common noun or else a list of adjectives followed by a common noun.

```
snouncla  = cnoun  ‘orelse‘ (structure (s1 adjs ++ s2 cnoun)
                            [a_rule 1(NOUNCLA_VAL‘of‘ lhs) EQ intrsct1
                                        [ADJ_VAL‘of‘s1,NOUNCLA_VAL‘of‘s2]])
```

The attribute rule `a_rule 1` states that the `NOUNCLA_VAL` value of the left-hand side of the syntax rule (i.e., the simple noun clause `snouncla`) is obtained by applying the semantic operator `intrsct1` to the `ADJ_VAL` of the list of adjectives `s1 adjs` with the `NOUNCLA_VAL` of the common noun `s2`.

The semantic functions are then defined as shown, where `intersect` is a predefined function. The database is also defined (within the program for prototyping or in external files). For example,

```
intrsct1 [ADJ_VAL x, NOUNCLA_VAL y]  = NOUNCLA_VAL (x `intersect` y)

set_of_moon = [Phobos, Deimos...]
orbit_rel   = [(Phobos, Mars), (Deimos, Mars) ...
```

Interpreters that are built in W/AGE are modular, and component evaluators can be used independently. For example,

```
snouncla (tokenize "red planet spins")
                       =>[[[NOUNCLA_VAL [Mars]],[WORD "spins"]]]
```

The W/AGE environment has been used to create natural language database-query interfaces which are hyperlinked in a Public-Domain SpeechWeb [Frost 2005] and which can be accessed through speech recognition interfaces running on remote lightweight end-user devices.

7.3. A Workbench for Experimenting with Montague-Like Grammars

Lapalme and Lavier [1990, 1993] developed a workbench in Miranda for experimenting with implementations of Montague-like approaches to natural language processing. Their implementation consists of four components which Montague claimed are necessary for a truth-conditional semantics:

First, a set of semantic values (these are entities, truth values, and functions constructed from them) are defined through a parameterized user-defined type:

```
data Semantic_value  a b
 = E a             -- entities          | FeFet (a -> a  -> b)
  | T b            -- truth values      | Ftt   (b -> b)
  | Fet   (a -> b) ...                  | FtFtt (b -> b -> b)  ...
```

The constructor `FeFet` indicates a function from an entity to a function from an entity to a truth value. The type `Semantic_value` can be instantiated for a specific set of individuals and truth values as illustrated in the following:

```
data People = Margaret | Elizabeth  | Robert etc.
Sem_people  = Semantic_value People Bool
```

Next, the type `Semantic_value` is parameterized so that different types of value can be used for entities and truth values. This seems a little odd at first but is used by Lapalme and Lavier to easily convert the interpreter to a processor which returns parse trees. For their example of an allocation stating the type of semantic value that is to be assigned to expressions of each syntactic category. Lapalme and Lavier choose the following assignment:

```
Cat. Constr. Type                        Cat. Constr. Type
N    E       People                      Conj FtFtt   Bool -> Bool -> Bool
Vi   Fet     People -> Bool              Neg  Ftt     Bool -> Bool
Vt   FeFet   People -> People -> Bool    S    T       Bool
```

Next, a set of semantic rules are defined, stating how the semantic values of composite expressions are computed from the semantic values of their components. Lapalme and Lavier define these rules in a single function `appff`:

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

The first line states that the result of applying a function `a` of type `People -> Bool` to a value `b` of type `People` is a value of type `Bool`, obtained by applying `a` to `b`.

Finally, semantic values are assigned to each of the basic expressions (words) in the language. For example,

```
f0 "Maggie" = E Margaret    f0 "sleeps" = Fet fs
f0 "Liz"    = E Elizabeth           where fs e = e == Margaret ||e == Robert
f0 "Bob"    = E Robert      f0 "and"    = FtFtt (&&)
```

The assignment states that `sleeps` denotes a function `fs` which returns `True` when applied to the entities `Margaret` or `Robert`, and `False` when applied to any other entity. This function, together with `appff`, is then integrated into a set of combinator parsers similar to those described in Section 5.3. The resulting processor `p` is such that, for example, `p "Liz sleeps and Bob sleeps" => T False`.

Lapalme and Lavier [1990, 1993] claim that Montague advocated for a clear separation between the semantic model and the syntactic analysis. They illustrate how their approach achieves this separation by simply changing the parameters of the type `Semantic_value` as follows: `Sem_tree = Semantic_value String_tree String_tree`, where `String_tree` is a user-defined type whose values are character-string representations of trees and redefining `f0` so that the semantic values assigned to words are `String-trees`. The values returned by the language processor are now `String-trees` which are transformed to a more readable form by a pretty-print function as shown in the examples that follow.

Lapalme and Lavier also show how variables can be incorporated into their processor in order to deal with ambiguity resulting from quantifier scoping in sentences such as "every man loves some woman". They begin by modifying the previous parser so that it translates the input to an intermediate form in which variables that range over the denotations of noun phrases become bound by quantifiers such as "every", "a", etc. For example,

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

This implements [Dowty et al. 1981, p. 69] where variables are introduced into the intermediate representation. For example, the representation of "every man snores" as `∀v1{man}, v1 snores`. The parser is further modified to generate all possible quantifier scopings, so that, for example,

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

Anaphoric sentences such as "Liz loves a man and that man sleeps" are accommodated by reference to the list of variables that has been created at the point that the word "that" is encountered. For example,

```
p  "Liz loves a man and that man sleeps"
 => variables ["man"]  @ for a v1
                          @@ N Liz
```

```
  VT loves
  v1
CONJ and
@ v1
  VI sleeps
```

Lapalme and Lavier [1990, 1993] recognize that their approach for dealing with quantifier scoping and anaphora is limited in its modeling of natural language. However, they claim that their examples illustrate the elegance with which the functional approach allows these features to be incorporated into a natural language interpreter.

7.4. Question-Answering (QA) and Information-Retrieval (IR) Systems

SATELITE is a natural language question-answering system which provides access to corporate information related to Telefonica de Espana, Madrid. It supports automatic spelling correction, ellipsis, and anaphora. The system, which is implemented in the lazy functional language NEL, was introduced in 1990 and, at one point, was responding to 50 queries a day. Apart from an entry on a Web page containing a list of applications of pure functional programming [Wadler 2005], there would appear to be no other publication describing this system.

Funser [1995] is a server for textual information retrieval from a 700-megabyte collection of full texts of French literature. Funser is implemented in the lazy functional programming language Alfonzo, which was specially developed for this application. At one time, Funser was accessed by over 500 users per month. The developers of Funser claim that LFP was found to be a powerful and elegant tool, but that performance results were mixed.

Rose et al. [2000] have developed a natural language interface for the retrieval of captioned images. The system, called ANVIL (Accurate Natural-language Visual Information Locator), uses a parser to extract information from the captions and the user query. The system uses WordNet synsets as a form of thesaurus. Terms in the captions and the user query are then expanded using the thesaurus and subsequently matched. An early prototype of the system was built in Haskell. Rose et al. state that the Haskell had a number of advantages as an implementation language specifically, the ability to quickly code complex algorithms, and the robust code which resulted from the powerful type system. However, the system was recoded in C++ because of a concern that the product development group, who did not have experience with Haskell, might not be able to provide long-term support.

Dalmas [2004] has Developed a Web question-answering system called Wee, and an associated question-answering model called QAAM, in Haskell. Web snippets are shallow parsed, filtered, and ranked using answer patterns. Repeated phrases are identified from the set of candidate sentences using a lazy longest-common-substring processor. Extracted phrases are used to generate an answer model which is then processed to discover relationships between the snippets from which the phrases were extracted. The resulting graph is further processed and the results presented to the user. The Wee system was entered at TREC-2004 [Ahn et al. 2004] as a stand-alone question-answering system and also in conjunction with the QED system which used deeper linguistic analysis and standard IR techniques. The results showed that Wee improved the results for factoid questions but not for definition questions.

7.5. Grammatical Framework

Grammatical Framework (GF) is a large multipurpose natural language processing system implemented in Haskell [Ranta 2004]. GF can be used to define grammars and

to perform various linguistic functions based on those grammars. Linguists can use GF to experiment with syntactic and semantic theories of language. Developers can use GF to create natural language processors of various kinds.

Central to GF is an abstract syntax which is based on type-theoretic grammar as described in Section 3.6. Users can manipulate abstract syntax trees using a structure editor (similar in some ways to the Cornel Synthesizer Generator). The abstract syntax trees are terms in a typed lambda calculus. Dependent types are used to represent semantic information such as gender, number, etc. A type-checker is used to determine agreement and other semantic properties.

Concrete syntax can be generated from the abstract syntax trees through a process of linearization. A single abstract syntax tree can be linearized in various ways generating output in different languages. GF also provides the ability to generate parsers from grammar definitions, allowing concrete syntax to be converted to one or more abstract syntax trees. Different types of parsers can be generated depending on the application.

One of the goals of GF is to help users to build natural language components on top of formal language processors, for example, GF provides that capability to build a German interface on top of a software-specification editor by separating the specification of formal problem-specific languages whose (sometimes complex mathematical) semantics are represented in the abstract syntax from the specification of the natural language features into which the abstract syntax is linearized. Experts in the problem domain write the application grammars, and linguists write the natural language grammars. The natural language grammars are called resource grammars and several have been built for GF, including English, Finnish, German, Italian, Russian, and Swedish [Khegai and Ranta 2004]. GF has been used in various applications. For example,

—multilingual document authoring. GF allows users to create and edit abstract syntax trees in one language using the syntax-directed editor, while seeing how the document evolves in another language. Amendments made to the document, such as changing the gender of the recipient of a letter, are then permeated throughout the tree(s) so that all translations are grammatically correct [Khegai et al. 2003].

—technical-document editing. GF has been used as a basis for a mathematical proof text editor [Hallgreen and Ranta 2000] and an XML editing tool [Dymetman et al. 2000].

—dialogue generation. The GF syntax-directed editor has also been used as the basis for a natural language dialogue system [Ranta and Cooper 2004].

—informal and formal requirements-specification tools [Hahnle et al. 2002] and translation from formal specifications to natural language [Burke and Johannisson 2005; Johannisson 2005].

GF can also be used for natural language translation and provides a solution to one of the major difficulties in this task, that is, the fact that the input source language may not contain all semantic information necessary to produce grammatically correct output in the target language. For example, when translating from English to German where more gender information is often required than is available in the English input, GF overcomes the problem by allowing the user to interact with the abstract syntax trees which are created as an intermediate representation during translation. The GF syntax-directed editor prompts the user to add additional semantic information as required during the process.

An embedded interpreter for GF and a compiler from GF grammars to speech recognition grammars have been implemented in Java by Bringert [2005]. The resulting system can be used to build multilingual dialog and translation systems for both spoken and written language.

8. USE OF LFP CONCEPTS IN NATURAL LANGUAGE ANALYSIS

So far, we have reviewed research that has involved the implementation of natural language processors in lazy functional programming languages. In addition to this, other researchers have considered how the principles and theories that are used to define and reason about lazy functional programming might provide insight into natural language.

8.1. Combinatory Parsing and Categorial Grammar

One of the motivations for Combinatory Categorial Grammar (Section 3.4) is the variable-free nature of the combinators which account for syntactic composition. It has been argued that the primitive operations of left and right application, composition, type-raising, etc. have more psycho-linguistic plausibility than mechanisms which involve variables. Bozsahin [1997] adds to this an argument that the associated semantics should also be based on combinatory logic. He makes reference to Turner [1979] in which a pure functional programming language is compiled into variable-free combinatory terms so as to obtain object code that can run more efficiently since it does not require environment creation and deletion. Bozsahin suggests that there could be an analogy with human cognitive processing.

Bozsahin uses his approach to explain word order variation in Turkish and argues that by representing the semantics as combinatory terms, the relationship between syntactic and semantic composition becomes easier to explain. As illustration, he shows how type-raising, together with the associated combinatory semantics, can be used to explain *scrambling* in Turkish (scrambling the variation in relative location of phrases denoting subjects, verbs, and objects) and claims that his approach provides a simple algebraic solution to word order variation. Although Bozsahin shows how his system can model various features of natural language, he states that extensive research on the cognitive aspects of the relationship between syntax and semantics needs to be done to support the hypothesis that natural language is intrinsically combinatory.

8.2. Monads and NL Theories

One of the difficulties in developing a comprehensive compositional semantic theory is that, as more complex aspects of natural language are added, the types and composition rules in the evolving theory have to be redefined. In some cases, all constructs have to be redefined even though only a few are affected by the added feature. According to Shan [2001a], Barbara Partee refers to this as "generalizing to the worst case". Shan has proposed a method to solve this problem by using monads to extend Montague-like semantic theories in order to accommodate additional aspects of language in a manner analogous to the use of monads to add additional computational capabilities to functional programs. To illustrate his proposed approach, Shan gives the following examples.

—A variation of the state monad [Wadler 1995] could be used to thread variable assignments through the evaluation process. Term phrases could be modified to update the variable assignment, and pronominals could be modified to refer to it. All other expressions could be upgraded by simple application of the unit function as they have no effect on variable assignment.

—A similar state monad could be used to thread representations of possible worlds through the evaluation process. Intensional expressions, such "the president" could be modified to refer to these worlds whereas words, such as "and" are not.

—The powerset monad could be used to accommodate ambiguity at the semantic level (rather than disambiguating the expression and then semantically processing the unambiguous forms).

8.3. Deep Types and Categorial Grammar

Deep types were developed in order to allow impure features to be added to pure functional programming languages in a systematic way. (*Shallow types* specify the program's functional type, whereas deep types are used to specify its behavior with respect to side effects). Korte [2004] has suggested that deep types could be added to Categorial Grammar in order to explain linguistic counterparts of side effects, such as intensionality, variable binding, quantification, interrogatives, focus or presupposition, as described by Shan [2003] in his paper on linguistic side effects.

8.4. Continuations in Natural Language

Continuations have been used for many years by computer scientists, particularly by functional programmers, as a tool for reasoning about control and order of evaluation and as an advanced programming construct. A continuation is an additional parameter which is given to a function and which is applied in that function's body to the result that would ordinarily be returned by the function. For example, consider the following simple function: `f x y = x + y` . Adding a continuation gives: `f' x y c = c (x + y)`, which can be read as the function `f'` adds its two arguments and then the computation continues with the function `c`, taking the result of this addition as argument. A number of advantages are claimed for the resulting continuation passing style (CPS) of programming: flow of control is made explicit, the identification of certain types of program transformation (e.g., to tail-recursive form) is facilitated, certain efficiencies can be obtained when values need not be returned through the stack of recursive function calls (e.g., in exception handling) and the ability to more easily compile a CPS program into efficient code. In addition, other Computer Scientists have used CPS to analyze programs and programming styles, for example, to model evaluation mechanisms such as call-by-name and to prove properties of programs which provide users with access to control flow as in the use of the back button in Web applications.

Shan and Barker [Shan 2001b; 2002; Barker 2002; Shan and Barker 2004] have investigated the use of continuations to explain a variety of linguistic features. Barker [2004] summarizes that work and gives, as examples focus quantifier ambiguity as in "everyone loved someone", in which emphasis is placed on one word in a sentence, for example, "JOHN saw Mary" compared with "John saw MARY", coordination in paraphrases, such as "John left and slept" and "John left and John slept", and misplaced modifiers, as in "John drank a quiet cup of tea". Barker notes that Montague also used a mechanism which is similar to a form of continuation-passing in his formal grammar.

As illustration of Shan and Barker's approach, we present a simplified description of an example given in Barker [2004] in which continuations are used to explain ambiguity in quantifier scoping. Consider the phrase "John saw everyone", which could be translated to `∀x saw(j, x)`. The word "everyone", which is embedded in the phrase "saw everyone", takes scope over the entire expression. Barker claims that continuations are useful in analyzing such phenomena as they have been used to provide formal descriptions of programming languages in which deeply embedded operators take control over enclosing expressions. Barker shows how continuation-based interpretation of ambiguous sentences such as "Someone saw everyone" can result in two denotations resulting from different transforms being applied corresponding to the left-to-right and right-to-left evaluation orders of the intermediate continuation-based interpretation

(cbi):

```
(Someone saw) everyone => cbi => transform => ∃x∀y (saw (x,y)
Someone (saw everyone) => cbi => transform => ∀y∃x (saw (x,y)
```

Other approaches do account for the two syntactic readings, but it appears that they require arbitrary manipulation of the intermediate forms to derive the two denotations. We have already referred in Section 3.3 to Pereira's criticism of Montague's approach in this respect. A similar criticism can be directed at the analysis described in Section 6.1, which does allow the interpretation of "Someone saw everyone" as both ((someone . saw) everyone) and (someone (saw everyone)).

Bozsahin, Shan, Korte and Barker appear to be motivated by the belief that the theoretical tools that are widely used for analyzing functional programs may have value in the analysis of natural language.

9. CONCLUDING COMMENTS

The research reviewed in this survey has shown that lazy functional programming can be used to

—Implement
(1) many of the parsers that are used by the linguistic community;
(2) highly-modular top-down combinator parsers which can accommodate ambiguous and left-recursive grammars in polynomial time and which are, therefore, ideally suited for for rapid prototyping of NLIs;
(3) more advanced combinator parsers that can accommodate various NL phenomena including dislocation, quantifier resolution, and some forms of context-sensitivity;
(4) robust type-directed parsers which can accommodate natural language expressions with ungrammatical word order.

—Encode
(1) direct representations of Montague-like compositional semantics for experimentation;
(2) efficient set-based versions of subsets of Montague semantics for use in natural language database-query processors;
(3) compositional semantic theories to accommodate complex phenomena such as dynamic quantifier scoping;
(4) large semantic nets for use in a variety of NL applications;
(5) type-theoretic semantics for use in the investigation of NL theories and implementation of NL applications;
(6) ontologies which can be automatically checked for errors.

—Construct
(1) large-scale NL systems based on semantic networks that can be used to build various applications including information extraction, foreign language tutoring, NL generation, metaphor processing, and discourse planning;
(2) executable attribute-grammar environments which can be used to build natural language database-query processors as executable specifications;
(3) frameworks for investigation of Montague-like compositional semantics;
(4) large-scale environments based on type-theoretic grammars that can be used to build various applications including multilingual document authoring, technical-document editing, dialogue generation, and NL translation.

In addition to demonstrating the use of LFP in NLI through the implementation of systems based on existing syntactic and semantic theories, other researchers have used

the LFP paradigm to investigate variations and extensions of these linguistic theories. For example, to accommodate negation under the closed-world assumption, to provide a uniform treatment of adjectives, to extend Montague semantics to accommodate transitive verbs with arity greater than two. Others have considered how the LFP paradigm might provide insight into natural language analysis. For example, in exploring the relationship between combinatory parsing and Categorial Grammar, the extension of Categorial Grammar with deep types, and the use of monads and continuations to explain complex linguistic phenomena.

Various claims have been made regarding the value of LFP in NLI.

—Value in syntactic analysis.
(1) The LFP stateless paradigm provides a useful framework by which similarities between various parsing strategies used in programming and natural language analysis can be made more clear.
(2) Implementation of conventional parsers benefits from the modularity, declarative nature, and lazy evaluation of LFP.
(3) Parser combinators, which are unique to LFP, allow language processors to be built as elegant programs whose structures are very similar to the grammars defining the languages to be processed. This facilitates implementation and experimentation with NLI design.
(4) The use of monads enables the construction of efficient parser combinators for ambiguous grammars and the accommodation of left-recursive productions, while maintaining the benefits of top-down search.
(5) The use of type classes facilitates the encoding and use of morphological specifications.

—Value in semantics.
(1) Many words in natural languages have denotations that are frequently defined by linguists as higher-order functions. The ability to define such functions directly in a LFP language and to pass them around as arguments facilitates the construction of NLIs.
(2) Polymorphism, user-defined types, and the strong type-checking provided by LFP languages facilitates the representation and investigation of semantic theories of natural language, including the specification and checking of ontologies.

—Value in the integration of syntactic and semantic analysis.
(1) Lazy evaluation allows semantic computation to be closely related to syntactic analysis without loss of efficiency. This is because lazy evaluation allows only those parts of the potentially huge parse space (corresponding to successful parses) to be evaluated by the semantic rules, resulting in modular and well-structured processors that can be used in real-time NLIs.
(1) The declarative nature of LFP languages, together with lazy evaluation, allows NLIs to be constructed piecewise as executable specifications of grammars which are themselves order-independent.

The LFP paradigm also has value for explaining natural language: Some researchers have argued that the theoretical tools that are used to analyze functional programs can also be used to analyze natural language due to the fact that natural language is inherently functional in nature.

Of course, such claims cannot be proven or disproven in any formal way. It is up to the reader to decide if the evidence and arguments presented in the surveyed papers substantiates them.

More needs to be done to determine the value of LFP in NLIs. More large-scale systems need to be built and experimental results analyzed. In addition, although

many of the researchers have stated that lazy evaluation facilitated their work, little explanation has been given. There is a need for a comprehensive theoretical study of the benefits of lazy evaluation in natural language processing.

10. ACKNOWLEDGMENTS

I would like to thank the anonymous reviewers for their constructive suggestions, those researchers who provided many useful and encouraging comments: Paul Callaghan, Joao Fernandes, Paul Hudak, John Hughes, Graham Hutton, Barbara Partee, David Turner, Jan van Eijck and Philip Wadler. In particular, Barbara Partee provided detailed comments which significantly improved the description of Montague Grammar, and Paul Callaghan was kind enough to give me a guided tour of Durham after an extensive review of the introductory notes on LFP, and a demonstration of the of the LOLITA system, the graduate students. Rahmatullah Hafiz, Fadi Hanna, and Nabil Abdullah, who helped with proofreading.

REFERENCES

ABDULLAH, N. 2003. Two set-theoretic approaches to the semantics of adjective-noun combinations. M.S. thesis, School of Computer Science, University of Windsor, Ontario, Canada.

ABDULLAH, N. AND FROST, R. A. 2005. Adjectives: A uniform semantic approach. In *Proceedings of Advances in Artificial Intelligence: the 18th Conference of the Canadian Society for Computational Studies of Intelligence (AI'02)*. B. Kegl and G. Lapalme, Eds. Lecture Notes in Computer Science, vol. 3501. Springer-Verlag, 330–341.

AHN, K., BOS, J., CLARK, S., CURRAN, J. R., DALMAS, T., LEIDNER, J. L., SMILLIE, M. B., AND WEBBER, B. 2004. Question answering with QED and WEE at TREC-2004. In *Proceedings of the 13th Text Retrieval Conference (TREC'04)*. E. M. Voorhees and L. P. Buckland, Eds. U.S. National Institute of Standards and Technology, (NIST), Gaithesburg, MD.

AJDUKIEWICZ, K. 1935. Die syntaktische konnexitat. *Studia Philosophica 1*, 1–27.

ANDERSSON, I. AND SODEREBERG, T. 2003. Spanish morphology implemented in a functional programming language. M.S. thesis, Department of Computing Science Chalmers University of Technology and the University of Gothenburg.

ANDROUTSOPOULOS, I., RITCHIE, G. D., AND THANISCH, P. 1995. Natural language interfaces to databases: An introduction. *J. Lang. Engin. 1*, 1, 29–81.

AUGUSTEIJN, L. 1990. The elegant compiler generator system. In *Proceedings of the International Conference WAGA: Attribute Grammars and their Applications*, P. Dransart and M. Jourdan, Eds. Lecture Notes in Computer Science, vol. 461. Springer-Verlag, 238–254.

BALDRIDGE, J. M. AND KRUIJFF, G. M. 2004. Course notes on combinatorial categorial grammar. http://esslli2004.loria.fr/content/readers/51.pdf.

BAR-HILLEL, Y. 1953. A quasi-arithmetical notation for syntactic description. *Language 29*, 47–58.

BARKER, C. 2002. Continuations and the nature of quantification. *Natural Language Seman. 10*, 211–242.

BARKER, C. 2004. Continuations in natural language. In *Proceedings of the 4th ACM SIGPLAN Continuations Workshop (CW'04)*, H. Thielecke, Ed. School of Computer Science, University of Birmingham, 1–11.

BENTHEM, J. V. 1986. Language in action: categories, lambdas and dynamic logic. *Sudies in Logic and the Foundation of Mathematics*. D. Reidel Publishing.

BENTHEM, J. V. 1991. Language in action: categories, lambdas and dynamic logic. *Sudies in Logic and the Foundation of Mathematics*, vol. 30. North-Holland.

BLACKBURN, P. AND BOS, J. 2005. *Representation and Inference for Natural Language. A First Course in Computational Semantics*. CSLI Publications, Stanford University.

BLACKBURN, P., DYMETMAN, M., LECOMTE, A., RANTA, A., RETORE, C., AND DE LA CLERGERIE, E. V. 1997. Logical aspects of computational linguistics: An introduction. In *Logical Aspects of Computational Linguistics*, C. Retore, Ed. Lecture Notes in Computer Science, vol. 1328. Springer-Verlag, 1–20.

BOGAVAC, L. 2004. Functional morphology for Russian. M.S. thesis, Department of Computing Science, Chalmers University of Technology and the University of Gothenburg.

BOSZAHIN, C. 1997. Combinatory logic and natural language parsing. *Elektrik, Turkish J. of EE and CS 5*, 3, 347–357.

BRINGET, B. 2005. Embedded grammars. M.S. thesis, Department of Computer Science and Engineering, Chalmers University of Technology and Gothenburg University.

BRUS, T., EEKELEN, M. V., LEER, M. V., PLASMEIJER, M. J., AND BARENDREGT, H. P. 1987. CLEAN—A language for functional graph rewriting. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture (FPCA'87)*, Kahn, Ed. Lecture Notes in Computer Science, vol. 274. Springer-Verlag, 364–384.

BURGE, W. H. 1975. *Recursive Programming Techniques*. Addison-Wesley Publishing Co., Reading, MA.

BURKE, D. A. AND JOHANNISSON, K. 2005. Translating formal software specifications to natural language/a grammar based approach. In *Proceedings of Logical Aspects of Computational Linguistics (LACL'05)*, P. Blace, E. Stabler, J. Busquets, and R. Moot, Eds. Lecture Notes in Artificial Intelligence, vol. 3402. Springer-Verlag, 52–66.

CALLAGHAN, P. C. 1998. An evaluation of LOLITA and related natural language processing systems. Ph.D. thesis, Department of Computer Science, University of Durham.

CALLAGHAN, P. C. 2005. Generalized LR parsing. In *The Happy User Guide* (Chap. 3). Simon Marlow.

CARPENTER, R. 1998. *Type-Logical Semantics*. Bradford Books.

CONSTANTINO, M. 1999. IE-Expert: Integrating natural language processing and expert systems techniques for real-time equity derivatives trading. *J. Computat. Intell. Finance 7*, 2, 34–52.

COPESTAKE, A. 2005. Natural language processing. Lecture Notes, Computer Laboratory, University of Cambridge.

CURRY, H. AND FEYS, R. 1958. Combinatory logic. *Studies in Logic*, vol. 1. North Holland.

DALMAS, T. 2004. *Wee/QAAM Manual*. School of Informatics, University of Edinburgh.

DOWTY, D. 1979. *Word Meaning and Montague Grammar*. D. Reidel Publishing Co.

DOWTY, D. R., WALL, R. E., AND PETERS, S. 1981. *Introduction to Montague Semantics*. D. Reidel Publishing Co.

DYMETMAN, M., LUX, V., AND RANTA, A. 2000. XML and multilingual document authoring: Convergent trends. In *Proceedings of the 18th International Conference on Computational Linguistics (COLING'00)*. Morgan Kaufmann, 243–249.

EIJCK, J. V. 2001. Incremental dynamics. *J. Logic, Language and Inform. 10*, 3, 319–351.

EIJCK, J. V. 2003. Parser combinators for extraction. In *Proceedings of the 14th Amsterdam Colloquium*, P. Dekker and R. van Rooy, Eds. 99–104.

EIJCK, J. V. 2004. Deductive parsing in haskell. Unpublished paper Uil-OTS/CWI/ILLC, Amsterdam and Utrecht.

EIJCK, J. V. AND NOUWEN, R. 2002. Quantification and reference in incremental processing. Unpublished paper, UiL-OTS/CWI/ILLC, Amsterdam and Utrecht.

FAIRBURN, J. 1986. Making form follow function: An exercise in functional programming style. Tech. rep. 89, Computer Laboratory, University of Cambridge.

FERNANDES, J. 2004. Generalized LR parsing in Haskell. Tech. rep. DI-PURe-04.11.01, Departamento de Informatica, da Universidade do Minho, Portugal.

FERNANDEZ, M. 1995. Spanish generation in the NL system LOLITA. M.S. thesis, Department of Computer Science, University of Durham.

FOKKER, J. 1995. Functional parsers. In *Advanced Functional Programming: 1st International Spring School on Advanced Functional Programming Techniques*, J. Jeuring and E. Meijer, Eds. Lecture Notes in Computer Science, vol. 924. Springer-Verlag, 1–23.

FORD, B. 2002. Packrat parsing: Simple, powerful, lazy, linear time. In *Procedings of the ACM SIGPLAN ICFP, International Conference on Functional Programming*. ACM Press, 36–47.

FORSBERG, M. 2004. Applications of functional programming in processing formal and natural languages. Licentiate thesis, Department of Computer Science and Engineering. Chalmers University of Technology and Gothenburg University.

FORSBERG, M. AND RANTA, A. 2004. Functional morphology. In *Procedings of the 9th ACM SIGPLAN International Conference on Functional Programming ICFP*. ACM Press, 213–223.

FRANK, A. U. 2001. Tiers of ontology and consistency constraints in geographical information systems. *Int. J. Geograph. Inform. Science 15*, 7, 667–678.

FROST, R. A. 1992. Constructing programs as executable attribute grammars. *The Comput. J. 35*, 4, 376–389.

FROST, R. A. 1993. Guarded attribute grammars. *Softw. Pract. Exper. 23*, 10, 1139–1156.

FROST, R. A. 2002. W/AGE the windsor attribute grammar programming environment. In *Proceedings of IEEE Symposia on Human Centric Computing Languages and Environments (HCC'02)*. 96–99.

FROST, R. A. 2003. Monadic memoization: Towards correctness-preserving reduction of search. In *Proceedings of Advances in Artificial Intelligence: 16th Conference of the Canadian Society for Computational Studies of Intelligence (AI'03)*, Y. Xiang and B. Chaib-draa, Eds. Lecture Notes in Artificial Intelligence, vol. 2671. Springer-Verlag, 66–80.

FROST, R. A. 2005. A call for a public-domain speechweb. *Comm. ACM 48*, 11, 45–49.

FROST, R. A. 2006. Functional pearl; polymorphism and the meaning of transitive verbs. Tech. rep. 06-006, School of Computer Science, University of Windsor, Ontario, Canada.

FROST, R. A. AND BOULOS, P. 2002. An efficient compositional semantics for natural language database queries with arbitrarily-nested quantification and negation. In *Proceedings of Advances in Artificial Intelligence: 15th Conference of the Canadian Society for Computational Studies of Intelligence (AI'02)*. R. Cohen and B. Spencer, Eds. Lecture Notes in Artificial Intelligence, vol. 2338. Springer-Verlag, 252–267.

FROST, R. A. AND HAFIZ, R. 2006. Using monads to accommodate ambiguity and left recursion with parser combinators. Tech. rep. 06-007, School of Computer Science, University of Windsor, Ontario, Canada.

FROST, R. A. AND LAUNCHBURY, E. J. 1989. Constructing natural language interpreters in a lazy functional language. *Comput. J*. (Special issue on Lazy Functional Programming) *32*, 2, 108–121.

FROST, R. A. AND SZYDLOWSKI, B. 1995. Memoizing purely-functional top-down backtracking language processors. *Science Comput. Program. 27*, 263–288.

GARIGLIANO, R., MORGAN, R., AND SMITH, M. 1992. LOLITA: Progress report 1. Tech. rep. 12/92, Department of Computer Science. University of Durham.

GARIGLIANO, R., MORGAN, R., AND SMITH, M. 1993. The LOLITA system as a contents scanning tool. In *Proceedings of the 13th International Conference on Artificial Intelligence, Expert Systems and Natural Language Processing*. Avignon, France.

GIRARD, J., LAFONT, Y., AND TAYLOR, P. 1988. Proofs and types. In *Cambridge Tracts in Theoretical Computer Science*, vol. 7. Cambridge University Press.

GRUNE, G. AND JACOBS, C. J. H. 1990. *Parsing Techniques; A Practical Guide*. Ellis Horwood, Chichester, England.

HAHNLE, R., JOHANNISSON, K., AND RANTA, A. 2002. An authoring tool for informal and formal requirements specifications. In *Proceedings of FASE Fundamental Approchaes to Software Engineering*. R. D. Kutsche and H. Weber, Eds. Lecture Notes in Computer Science, vol. 2306. Springer-Verlag, 233–248.

HALLGREEN, T. AND RANTA, A. 2000. An extensible proof text editor. In *Proceedings of (LPAR'00)*. M. Parigot and A. Voronkov, Eds. Lecture Notes in Artificial Intelligence, vol. 1955. Springer-Verlag, 70–84.

HEITZ, J. 1996. An investigation into figurative language in the LOLITA NLP system. M.S. thesis, Department of Computer Science, University of Durham.

HENDRIKS, H. 1993. Studied flexibility: Categories and types in syntax and semantics. Ph.D. thesis, Universiteit van Amsterdam.

HILL, S. 1996. Combinators for parsing expressions. *J. Funct. Program. 6*, 3, 445–463.

HINRICHS, E. W. 1988. Tense, quantifiers, and contexts. *Computat. Linguist. 14*, 2, 3–14.

HOPCROFT, J. E., ULLMAN, J. D., AND MOTWANI, R. 2000. *Introduction to Automata Theory, Languages, and Computation*, 2nd Ed. Addison Wesley.

HUDAK, P., PETERSON, J., AND FASEL, J. 2000. A gentle introduction to Haskell. www.Haskell.org.

HUDAK, P., PEYTON-JONES, S. L., WADLER, P., BOUTEL, B., FAIRBAIRN, J., FASEL, J. H., GUZMAN, M. M., HAMMOND, K., HUGHES, J., JOHNSSON, T., KIERBURTZ, R. B., NIKHIL, R. S., PARTAIN, W., AND PETERSON, J. 1992. Report on the programming language Haskell, a non-strict, purely functional language. *SIGPLAN Notices 27*, 5, R1–R164.

HUET, G. 2003. Zen and the art of symbolic computing: Light and fast applicative algorithms for computational linguistics. In *Proceedings of Practical Aspects of Declarative Languages Symposium (PADL'03)*, V. Dahl and P. Wadler, Eds. Lecture Notes in Artificial Intelligence, vol. 2562. Springer-Verlag, 252–267.

HUET, G. 2005. Transducers as lexicon morphisms, phonemic segmentation by euphony analysis, application to a sanskrit tagger. *J. Funct. Program. 15*, 4, 573–614.

HUGHES, R. J. M. 1989. Why functional programming matters. *Comput. J.* (Special Issue on Lazy Functional Programming). *32*, 2, 98–107.

HUGHES, R. J. M. 2000. Generalizing monads to arrows. *Science Comp. Program. 37*, 67–111.

HUTTON, G. 1992. Higher-order functions for parsing. *J. Funct. Program. 2*, 3, 323–343.

HUTTON, G. AND MEIJER, E. 1998. Monadic parser combinators. *J. Funct. Program. 8*, 4, 437–444.

IWANSKA, L. 1992. A general semantic model of negation in natural language: Representation and inference. Ph.D. thesis, Computer Science, University of Illinois at Urbana-Champaign.

JEURING, J. AND SWIERSTRA, S. D. 1994. Bottom-up grammar analysis. In *Proceedings of Programming Languages and Systems, (ESOP'94)*, D. Sannella, Ed. Lecture Notes in Computer Science, vol. 788. Springer-Verlag, 317–332.

JEURING, J. AND SWIERSTRA, S. D. 1995. Constructing functional programs for grammar analysis problems. In *Proceedings of Conference Record of FPCA'95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*. 259–269.

JOHANNISSON, K. 2005. Formal and informal software specifications. Ph.D. thesis, Department of Computer Science and Engineering. Chalmers University of Technology and Gothenburg University.

JOHNSON, M. 1995. Squibs and discussions: Memoization in top-down parsing. *Computat. Linguist 21*, 3, 405–417.

JOHNSSON, T. 1987. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture*, G. Kahn, Ed. Lecture Notes in Computer Science, vol. 274. Springer-Verlag, 154–173.

JONES, M. P. 1992. Introduction to gofer 2.20. Tech. rep., Programming Research Group, Oxford University.

JONES, M. P., HUDAK, P., AND SHUAMYAN, S. 1995. Using types to parse natural language. In *Proceedings of the Glasgow Workshop on Functional Programming*. Workshops in Computer Science Series. (IFIP), Springer-Verlag.

KHEGAI, J., NORDSTRM, B., AND RANTA, A. 2003. Multilingual syntax editing in GF. In *Proceedings of the 4th International Conference on Intelligent Text Processing and Computational Linguistics (CICLing'03)*, A. F. Gelbukh, Ed. Lecture Notes in Computer Science, vol. 2588. Springer-Verlag, 453–464.

KHEGAI, J. AND RANTA, A. 2004. Building and using a Russian resource grammar in GF. In *Proceedings of the 5th International Conference on Intelligent Text Processing and Computational Linguistics (CICLing'04)*, A. F. Gelbukh, Ed. Lecture Notes in Computer Science, vol. 2945. Springer-Verlag, 38–41.

KOOPMAN, P. AND PLASMEIJER, R. 1999. Efficient combinator parsers. In *Proceedings of Implementation of Functional Languages: Tenth International Workshop (IFL'98)*. K. Hammand, T. Davie, and C. Clack, Eds. Lecture Notes in Computer Science, vol. 1595. Springer-Verlag, 122–138.

KORTE, L. 2004. Deep types for categorial grammar: A side effect analysis. In *Proceedings of TAAL Postgraduate Conference*. Edinburgh University.

KUDLEK, M., MARTIN-VIDE, C., MATEESCU, A., AND MITRANA, V. 2003. Contexts and the concept of mild-context-sensitivity. *Linguist. Philos. 26*, 703–725.

KUHN, W. 2002. Modelling the semantics of geographical categories through conceptual integration. In *GIScience*. Lecture Notes in Computer Science, vol. 2478. Springer-Verlag, 108–118.

KUNO, S. 1965. The predictive analyzer and a path elimination technique. *Comm. ACM 8*, 7, 453–462.

LAMBEK, J. 1958. The mathematics of sentence structure. *Amer. Mathemat. Month. 65*, 154–170.

LAPALME, G. AND LAVIER, F. 1990. Using a functional language for parsing and semantic processing. Tech. rep. 715a, Departement d'informatique et recherche operationelle, Universite de Montreal.

LAPALME, G. AND LAVIER, F. 1993. Using a functional language for parsing and semantic processing. *Computat. Intell. 9*, 111–131.

LEERMAKERS, R. 1993. *The Functional Treatment of Parsing*. International Series in Engineering and Computer Science. Kluwar Academic Publishers.

LEIJEN, D. AND MEIJER, E. 2001. Parsec: Direct style monadic parser combinators for the real world. Tech. rep. UU-CS-2001-35, Department of Computer Science, University of Utrecht.

LICKMAN, P. 1995. Parsing with fixed points. M.S. thesis, University of Cambridge.

LJUNGLOF, P. 2002a. Functional programming and NLP. Tech. rep., Department of Computer Science, Chalmers University.

LJUNGLOF, P. 2002b. Pure functional parsing-an advanced tutorial. Licentiate thesis, Department of Computing Science, University of Gothenburg.

LJUNGLOF, P. 2004. Functional pearls: Functional chart parsing of context-free grammars. functional pearl. *J. Funct. Program. 14*, 6, 669–680.

LOIDL, H., MORGAN, R., TRINDER, P., PORIA, S., COOPER, C., PEYTON-JONES, S., AND GARIGLIANO, R. 1997. Parallelising a large functional program, or: keeping LOLITA busy. In *International Workshop on the Implementation of Functional Languages*, C. Clack, K. Hammond, and T. Davie, Eds. Lecture Notes in Computer Science, vol. 1467. Springer-Verlag, 198–213.

LONG, D. AND GARIGLIANO, R. 1994. *Analogy and Causality (A Model and Application)*. Ellis Horwood, Chichester, UK.

MARLOW, S. 2005. *The Happy User Guide*. http://www.Haskell.org/happy/doc/html/index.html.

MARTIN-LOF, P. 1980. Intuitionistic type theory. Notes by Giovani Sambin of a series of lectures given in Padua, Bibliopolis, Napoli.

MEDLOCK, B. 2002. A tool for generalized LR parsing in Haskell. Single honours C.S. project report, Department of Computer Science, University of Durham.

MEZIANE, F. AND METAIS, E., Eds. 2004. *Natural Language Processing and Information Systems: 9th International Conference on Applications of Natural Language to Information Systems, (NLDB'04)*. Lecture Notes in Computer Science, vol. 3136. Springer-Verlag.

MONTAGUE, R. 1970. Universal grammar. *Theoria 36*, 373–398. (Reprinted in Thomason 1974, 222–246.)

MONTAGUE, R. 1973. The proper treatment of quantification in ordinary English. In *Approaches to Natural Language*, K. J. J. Hintikka, J. M. E. Moravcsik, and P. Suppes, Eds. D. Reidel Publishing Co., 221–242.

MOORTGAT, M. 1988. *Categorial Investigations. Logical and Linguistic Aspects of the Lambek Calculus*. Foris Publications, Dordrecht.

MORGAN, K., GARIGLIANO, R., CALLAGHAN, P., PORIA, S., SMITH, M., URBANOWICZ, A., COLLINGHAM, R., CONSTANTINO, M., COOPER, C., AND THE LOLITA GROUP, UNIVERSITY OF DURHAM, U. 1995. Description of the LOLITA system as used for MUC-6. In *Proceedings of the 6th Message Understanding Conference (MUC6)*. NIST, Morgan-Kaufmann.

NIKHIL, R. S. 1993. A multithreaded implementation of Id using P-RISC graphs. In *Proceedings of Languages and Compilers for Parallel Computing LCPC , 6th International Workshop*, U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua, Eds. Lecture Notes in Computer Science, vol. 768. Springer-Verlag (1994), 390–405.

NORVIG, P. 1991. Techniques for automatic memoisation with applications to context-free parsing. *Computat. Linguist. 17*, 1, 91–98.

PAAKI, J. 1995. Attribute grammar paradigms—A high-level methodology in language implementation. *ACM Comput. Surv. 27*, 2, 196–256.

PACE, G. 2004. Monadic compositional parsing with context using maltese as a case study. In *Proceedings of the Computer Science Annual Workshop (CSAW'04)*, University of Malta, G. Pace and J. Cordina, Eds. 60–70.

PANITZ, S. E. 1996. Termination proofs for a lazy functional language by abstract reduction. Tech. rep. 06, J. W. Goethe-Universitat. citeseer.nj.nec.com/panitz96termination.html.

PARTEE, B. H. 1975. Montague grammar and transformational grammar. *Linguis. Inquiry 6*, 2, 203–300.

PARTEE, B. H., Ed. 1976. *Montague Grammar*. Academic Press, New York, NY.

PARTEE, B. H. 2001. Montague Grammar. In *International Encyclopedia of the Social and Behavioral Sciences*, N. J. Smelser and P. B. Baltes, Eds. Elsevier.

PARTEE, B. H. AND HENDRICKS, L. W. 1997. Montague Grammar. In *Handbook of Logic and Language*, J. van Benthem and A. ter Meulen, Eds. Elsevier, 5–91.

PARTRIDGE, A. AND WRIGHT, D. 1996. Predictive parser combinators need four values to report errors. *J. Funct. Program. 6*, 2, 355–364.

PEMBECCI, I. 1995. A combinator parser for the morphological analysis of Turkish. Senior project report, Department of Computer Engineering, Middle East Technical University, Ankara.

PEYTON-JONES, S. 2003. The Haskell 98 language. *J. Funct. Program. 13*, 1, 0–255.

RANTA, A. 1994. *Type-Theoretical Grammar*. Oxford University Press, Oxford, UK.

RANTA, A. 1995. Type-theoretical interpretation and generalization of phrase structure grammar. *Bull. of the IGPL 3*, 2, 319–342.

RANTA, A. 2001. 1+n representations of Italian morphology. Essays dedicated to Jan von Plato on the occasion of his 50th birthday.

RANTA, A. 2004. Grammatical framework. *J. Funct. Program. 14*, 2, 145–189.

RANTA, A. AND COOPER, R. 2004. Dialogue systems as proof editors. *J. Logic, Language Inform. 13*, 2, 225–240.

REED, C., LONG, D., FOX, M., AND GARAGNANI, M. 1997. Persuasion as a form of inter-agent negotiation. In *Proceedings of Workshop on Distributed Artificial Intelligence (PRICAI'96)*. Lecture Notes in Computer Science, vol. 1286. Springer-Verlag, 120–136.

ROCHE, E. AND SCHABES, Y. 1997. *Finite-State Language Processing*. Bradford Books.

ROSE, T., ELWORTHY, D., KOTCHE, A., CLARE, A., AND TSONIS, P. 2000. ANVIL: A system for the retrieval of captioned images using NLP techniques. In *Proceedings of Challenge of Image Retrieval (CIR'00)*. J. P. Eakins and P. G. B. Enser, Eds. University of Brighton, UK.

ROY, M. 2005. Extending a set-theoretic implementation of Montague Semantics to accommodate n-ary trnasitive verbs. M.S. thesis, School of Computer Science, University of Windsor, Ontario, Canada.

ROY, M. AND FROST, R. 2004. Extending Montague Sematics for use in natural language database-query processing. In *Proceedings of Advances in Artificial Intelligence: The 17th Conference of the Canadian Society for Computational Studies of Intelligences (AI'04)*. A. Tawfik and S. Goodwin, Eds. Lecture Notes in Computer Science, vol. 3060. Springer-Verlag, 567–568.

SAVITCH, W. J. 1989. A formal model for context-free languages augmented with reduplication. *Computat. Linguist. 15*, 4, 250–261.

SHAN, C. 2001a. Monads for natural language semantics. In *Proceedings of the 13th European Summer School in Logic, Language and Information.Student Session (ESSLLI'01)*, K. Striegnitz, Ed. Helsinki, Finland, 285–298.

SHAN, C. 2001b. A variable-free dynamic semantics. In *Proceedings of the 13th Amsterdam Colloquium*, R. van Rooy and M. Stokhof, Eds. Institute for Logic, Language and Computation, Universiteit van Amsterdam, 204–209.

SHAN, C. 2002. A continuation semantics of interrogatives that accounts for baker's ambiguity. In *Semantics and Linguistic Theory (SALT XII)*. B. Jackson, Ed. Cornell University Press, 246–265.

SHAN, C. 2003. Linguistic side effects. In *Proceedings of the 18th Annual IEEE Symposium on Logic and Computer Science (LICS'03) Workshop on Logic and Computational Linguistics*. L. Libkin and G. Penn, Eds. Ottawa, Canada.

SHAN, C. AND BARKER, C. 2004. Explaining crossover and superiority as left-to-right evaluation. In *Worshop on Semantic Approaches to Binding Theory (ESSLLI'04), the 16th European Summer School in Logic, Language and Information*. E. Keenan and P. Schlenker, Eds. Nancy, France.

SHAUMYAN, S. 1977. *Applicational Grammar as a Semantic Theory of Natural Language*. Edinburgh University Press.

SHAUMYAN, S. AND HUDAK, P. 1997. Linguistic, philosophical, and pragmatics aspects of type-directed natural langugae. In *Proceedings of Logical Aspects of Computational Linguistics: 2nd International Conference (LACL'97)*. A. Lecomte, F. Lamarche, and G. Perrier, Eds. Lecture Notes in Computer Science, vol. 1582. Springer-Verlag, 70–91.

SHAUMYAN, S. AND SEGOND, F. 1994. Long-distance dependencies and applicative universal grammar. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING)*. Kyoto, Japan, 853–858.

SHIEBER, S. M. 1985. Evidence against the context-freeness of natural language. *Linguist. Philos. 8*, 333–343.

SHIEL, B. A. 1976. Observations on context-free parsing. Tech. rep. TR 12-76, Center for Research in Computing Technology, Aiken Computational Laboratory, Harvard University.

SHIU, S., LUO, Z., AND GARIGLIANO, R. 1996. Type theoretic semantics for SemNet. In *Practical Reasoning: International Conference on Formal and Applied Practical Reasoning (FAPR'96)*. D. Gabbay and H. J. Ohlbach, Eds. Lecture Notes in Artificial Intelligence, vol. 1085. Springer-Verlag, 582–595.

SHIU, S. K. Y. 1997. Type theoretic semantics for semantic networks: An application to natural language engineering. Ph.D. thesis, Department of Computer Science, University of Durham.

SHORT, S., SHIU, S., AND GARIGLIANO, R. 1996. Distributedness and non-linearity of LOLITA's semantic network. In *Proceedings of the 16th International Conference on Computational Linguistics*. Center for Sprogteknologi, Copenhagen, 436–441.

SMITH, M. H. 1996. Natural language generation in the LOLITA system: An engineering approach. Ph.D. thesis, Department of Computer Science, University of Durham.

SMITH, M. H., GARIGLIANO, R., AND MORGAN, R. 1994. Generation in the LOLITA system: An engineering approach. In *Proceedings of the 16th International Natural Language Generation Workshop*. 241–244.

STEEDMAN, M. 1991. Type-raising and directionality in combinatory grammar. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics (ACL)*. Berkeley CA, 71–79.

STEEDMAN, M. 1996. A very short introduction to CCG. Unpublished paper. http://www.coqsci.ed.ac.uk/steedman/paper.html

STEEDMAN, M. 1999. Alternating quantifier scope in CCG. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL)*. Morgan Kaufmann, 301–308.

STEEDMAN, M. AND BALDRIDGE, J. 2003. Combinatory categorial grammar. Unpublished tutorial, School of Informatics, Edinburgh University. ftp://ftp.cogsci.ed.ac.uk/pub/steedman/ccg/manifesto.pdf.

STOY, J. E. 1977. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA.

SWIERSTRA, S. AND DUPONCHEEL, L. 1996. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programmin*, J. Launchbury, E. Meijer, and T. Sheard, Eds. Lecture Notes in Computer Science, vol. 1129. Springer-Verlag, 184–207.

SYPNIEWSKI, B. P. 1999. An introduction to applicative universal grammar. Unpublished paper. http://elvis.rowan.edu/ bps/ling/introAUG.pdf.

SZYDLOWSKI, B. 1996. Complexity analysis and monadic specification of memoized functional parsers. M.S. thesis, School of Computer Science, University of Windsor.

THOMASON, R. H. 1974. *Formal Philosophy: Selected Papers of Richard Montague*. Yale University Press, New Haven CT.

TOMITA, M. 1985. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers.

TURNER, D. A. 1979. A new implementation technique for applicative languages. *Softw. Pract. Exper. 9*, 1, 31–49.

TURNER, D. A. 1981. Aspects of the implementation of programming languages. Ph.D. thesis, Oxford University, Oxford, UK.

TURNER, D. A. 1985. Miranda: a lazy functional programming language with polymorphic types. In *Proceedings of the IFIP International Conference on Functional Programmiong Languages and Computer Architecture*. J. Jouannaud, Ed. Lecture Notes in Computer Science, vol. 201. Springer Verlag, 1–16.

TURNER, D. A. 1986. An overview of miranda. *SIGPLAN Notices 21*, 12, 158–166.

UDDERBORG, G. 1988. A functional parser generator. Licentiate thesis, Chalmers University of Technology, Gothenburg.

VAN BENTHEM, V. 1987. Categorial grammars and lambda calculus. In *Mathematical Logic and its Applications*, D. Skordev, Ed. Plenum Press.

WADLER, P. J. 1985. How to replace failure by a list of successes. In *Proceedings of the IFIP International Conference on Functional Programmiong Languages and Computer Architecture*, J. Jouannaud, Ed. Lecture Notes in Computer Science, vol. 201. Springer-Verlag, 113–128.

WADLER, P. J. 1989. Special edition on lazy functional programming. *Comput. J. 32*, 2.

WADLER, P. J. 1990. Comprehending monads. In *Proceedings of the ACM SIGPLAN/SIGACT/SIGART Symposium on Lisp and Functional Programming*. ACM Press, 61–78.

WADLER, P. J. 1994. Tech. rep., http://homepages.inf.ed.ac.uk/wadler/realworld/satelite.html.

WADLER, P. J. 1995. Monads for functional programming. In *1st International Spring School on Advanced Functional Programming Techniques*, J. Jeuring and E. Meijer, Eds. Lecture Notes in Computer Science, vol. 924. Springer-Verlag, 24–52.

WANG, Y. 1994. An intelligent computer-based tutoring approach for the management of negative transfer. Ph.D. thesis, Department of Computer Science, Durham University.

ZIFF, D. A., SPACKMAN, S. P., AND WACLENA, K. 1995. Funser: A functional server for textual information retrieval. *J. Funct. Program. 5*, 3, 317–343.

Received July 2005; revised May 2006; accepted May 2006