

# Lazy Combinators for Executable Specifications of General Attribute Grammars

Rahmatullah Hafiz and Richard A. Frost

School of Computer Science, University of Windsor  
401 Sunset Avenue Windsor, ON N9B 3P4 Canada  
{hafiz,richard}@uwindsor.ca

**Abstract.** A lazy-evaluation based top-down parsing algorithm has been implemented as a set of higher-order functions (combinators) which support directly-executable specifications of fully general attribute grammars. This approach extends aspects of previous approaches, and allows natural language processors to be constructed as modular and declarative specifications while accommodating ambiguous context-free grammars (including direct and indirect left-recursive rules), augmented with semantic rules with arbitrary attribute dependencies (including dependencies from right). This one-pass syntactic and semantic analysis method has polynomial time and space (w.r.t. the input length) for processing ambiguous input, and helps language developers build and test their models with little concern for the underlying computational methods.

**Keywords :**Parser combinators, Lazy evaluation, Top-down parsing, Attribute grammars, Natural-language processing

## 1 Introduction

Attribute grammar (AG, [1]) systems have been constructed primarily as compilable parser-generators for formal languages. Little work has been done where fully-general AGs have been used to offer a platform for declaratively specifying directly-executable specifications of natural languages (NL) to construct NL interfaces or NL database query processors. Although it is highly modular, general top-down parsing is often ignored as it has been traditionally categorized as expensive, and non-terminating while processing left-recursive grammars. Also, no existing approach supports arbitrary attribute dependencies (including dependencies from the right) in one-pass within a modular top-down system.

A platform that supports executable and declarative specifications of general AGs, offers two benefits. From a practical viewpoint, application developers can specify and execute their language descriptions directly without worrying about underlying evaluation methods. Individual parts of descriptions can be efficiently tested piecewise, and modularity enables systematic and incremental development. From a theoretical perspective, general AGs accommodate ambiguity and left-recursion, which are needed for natural language processing. As illustrated by Warren [2] and Frost et al. [3], transforming a left-recursive CFG to a weakly

equivalent non-left-recursive form may introduce loss of parses and lack of completeness in semantic interpretation. AGs with arbitrary attribute dependencies provide unrestricted construction of declarative semantic rules, facilitating expression of complex linguistic theories such as Montague semantics.

Frost et al. explained at PADL'08 [4] how top-down parsers can be constructed as unconstrained executable CFGs. This paper describes an extension to accommodate semantics with arbitrary attribute dependencies. We have achieved our objective by defining a set of combinators for constructing modular, declarative and executable language processors, similar to the denotational semantics textbook AG notation. Our combinators (e.g., `<|>` and `*>` correspond to alternating and sequencing, `rule_s` and `rule_i` for synthesized and inherited semantic rules, `parser` and `nt` for AG formation) are pure, higher-order and lazy functions that ensure fully declarative specifications (Section 3.2 and 3.3).

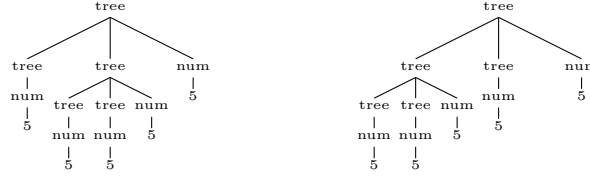
We define attributes in terms of *expressions* (as our method is referentially transparent and non-strict) that represent operations on syntax symbols, and these expressions are computed from the surrounding *environment* when required (Section 3.4). We execute syntax and semantics in polynomial time using memoization to ensure that results for a particular parser at a particular input position are computed at most once, and are reused when required. We represent potentially exponential results for ambiguous input in a compact and shared polynomial tree structure (Section 4).

We have provided a platform by implementing our algorithm in terms of higher-order functions in Haskell. The declarative notation, arbitrary dependencies and non-strict evaluation have the potential to allow us to discard unwanted parses using linguistic features such as grammatical, semantic and number agreements, and this could extend the AG paradigm by capturing characteristics of unification grammars, combinatory-categorical grammars and type-theoretic grammars while being computationally efficient.

**An Example:** We illustrate our approach with a simple artificial `repmax` example [5,6] which we have extended to accommodate ambiguity, left-recursion and arbitrary attribute dependencies in semantic rules. Our goal is to parse inputs such as “1 5 2 3 2” with the ambiguous left-recursive CFG  $tree ::= tree\ tree\ num \mid num$ ,  $num ::= 1|2|3|4|5|...$ , and to extract all possible trees with all terminals replaced by the maximum value of the sequence using sets of declarative semantics. The following example illustrates most of the aspects of general AGs :

$$\begin{aligned}
 start(S_0) &:: = tree(T_0) \\
 &\{RepVal.T_0 \downarrow = MaxVal.T_0 \uparrow\} \\
 tree(T_0) &:: = tree(T_1)\ tree(T_2)\ num(N_1) \\
 &\{MaxVal.T_0 \uparrow = Max(MaxVal.T_1 \uparrow, MaxVal.T_2 \uparrow, MaxVal.N_1 \uparrow), \\
 &\quad RepVal.T_1 \downarrow = RepVal.T_0 \downarrow, RepVal.T_2 \downarrow = RepVal.T_0 \downarrow, \\
 &\quad RepVal.N_1 \downarrow = RepVal.T_0 \downarrow\} \\
 &\quad | \quad num(N_2) \\
 &\{MaxVal.T_0 \uparrow = MaxVal.N_2 \uparrow, RepVal.N_2 \downarrow = RepVal.T_0 \downarrow\} \\
 num(N_0) &:: = 1 \{MaxVal.N_0 \uparrow = 1\} | \dots | 5 \{MaxVal.N_0 \uparrow = 5\}
 \end{aligned}$$

According to this AG, there are two ambiguous outputs when *start* is applied to the input sequence “1 5 2 3 2”:



Using our method, an “almost verbatim” executable specification of the above AG’s representation can be constructed in Haskell as follows:

```
start = memoize Start parser (nt tree T0)
      [rule_i RepVal Of T0 Is findRep [synthesized MaxVal Of T0]]

tree = memoize Tree parser
      (nt tree T1 *> nt tree T2 *> nt num T3)
      [rule_s MaxVal Of LHS Is
        findMax [synthesized MaxVal Of T1,synthesized MaxVal Of T2,synthesized MaxVal Of T3]
      ,rule_i RepVal Of T1 Is findRep [inherited RepVal Of LHS]
      ,.....
      <|> parser (nt num N1)
      [rule_i RepVal Of N1 Is findRep [inherited RepVal Of LHS]
      ,rule_s MaxVal Of LHS Is findMax [synthesized MaxVal Of N1]]

num = memoize Num terminal term "1" [MaxVal 1] <|> ... <|> terminal term "5" [MaxVal 5]
```

When the executable specification of *start* is applied to “1 5 2 3 2”, a compact representation of ambiguous parse trees is generated with appropriate semantic values for respective grammar symbols. For example, *tree* parses the whole input (starting at position 1 and ending at position 6) in two ambiguous ways. The *tree*’s inherited and synthesized attributes (represented with I and S) are associated with its start and end positions respectively. The attributes are of the form *attribute\_type* value e.g. *RepVal* 5. The compact results have pointing sub-nodes (as node-name, unique-id pairs e.g. (*Tree*,T1)) with inherited and synthesized attributes:

```
Tree START at 1 ; Inherited atts:  T0 RepVal 5
  END at 6 ; Synthesized atts: T0 MaxVal 5
Branch
[SubNode (Tree,T1) ((1,[(I,T1),[RepVal 5]]),(4,[(S,T1),[MaxVal 5]]))]
,SubNode (Tree,T2) ((4,[(I,T2),[RepVal 5]]),(5,[(S,T2),[MaxVal 3]]))]
,SubNode (Num, T3) ((5,[(I,T3),[RepVal 5]]),(6,[(S,T3),[MaxVal 2]]))]
  END at 6 ; Synthesized atts: T0 MaxVal 5
Branch
[SubNode (Tree,T1) ((1,[(I,T1),[RepVal 5]]),(2,[(S,T1),[MaxVal 1]]))]
,SubNode (Tree,T2) ((2,[(I,T2),[RepVal 5]]),(5,[(S,T2),[MaxVal 5]]))]
,SubNode (Num, T3) ((5,[(I,T3),[RepVal 5]]),(6,[(S,T3),[MaxVal 2]]))]
  .....
Num START at 1 ; Inherited atts:  N1 RepVal 5
  END at 2 ; Synthesized atts: N1 MaxVal 1
  Leaf (ALeaf "1", (S,N1)).....
  START at 5 ; Inherited atts:  N1 RepVal 5
  END at 6 ; Synthesized atts: N1 MaxVal 2
  Leaf (ALeaf "2", (S,N1))
```

This example illustrates that complex semantic rules can be accommodated. Our semantic rules declaratively define arbitrary actions on the syntax symbols.

For example, the second semantic rule of `tree` is an inherited rule for the second parser `T2`, which depends on its ancestor `T0`'s inherited attribute `RepVal`. The `T0`'s `RepVal` is dependent on its own synthesized attribute `MaxVal`, and eventually this `RepVal` is threaded down as every `num`'s inherited attribute.

## 2 General AGs and Parser Combinators

In an attribute grammar, syntax rules of a context-free grammar are augmented with semantic rules to describe the meaning of the sentences of a context-free language. Although different definitions have been given [6–8, etc.], we prefer to define a general AG by imposing minimal restrictions on attribute dependencies: a CFG is 4-tuple  $G = (N, T, P, S)$ , where  $N$  is a finite set of non-terminals,  $T$  is a finite-set of terminals,  $P$  is a finite-set of syntax rules,  $S$  is the start non-terminal,  $N \cap T = \phi$  and  $(\forall p_i \in P) p_i$  is of the form  $a ::= b$  where  $a \in N$  and  $b \in (N \cup T)^*$ .

An AG can be formed from  $G$  as a 3-tuple  $AG = (G, A, R)$ , where  $A$  is a finite set of attributes and  $R$  is a finite set of semantic rules. Each  $X \in (N \cup T)$  is associated with a set of attributes  $A(X) \subset A$ , and each  $a \in A(X)$  can be described by a function  $r \in R$ . The set  $A(X)$  can be partitioned into two sets  $A_i(X)$  and  $A_s(X)$ , which represents *inherited* and *synthesized* attributes respectively. A synthesized attribute is an attribute for the LHS symbol of a production rule, and an inherited attribute is associated with a symbol that resides at the RHS of the production rule. We define the inherited and synthesized expressions  $r_{a_i}$  and  $r_{a_s}$  (w.r.t. a syntax rule  $X_0 ::= X_1 X_2 \dots X_n$ ), that generate each  $a_i \in A_i(X)$  and  $a_s \in A_s(X)$  respectively as:

$$\begin{aligned}
 r_{a_i} : \mathbf{P}(\bigcup A(X)) &\rightarrow A_i(X_x) \\
 &\alpha \mapsto \text{operations on } \alpha \\
 r_{a_s} : \mathbf{P}(\bigcup A(X) - A(X_0)) &\rightarrow A_s(X_0) \\
 &\alpha \mapsto \text{operations on } \alpha \\
 &\text{where } 0 < x \leq n \text{ and } \mathbf{P} \text{ is the power set}
 \end{aligned}$$

In functional programming, parser combinators have been used extensively [9–11, etc.] to prototype top-down backtracking recognizers, which provide modular and executable specifications of grammars that accommodate ambiguity. In basic recursive-descent top-down recognition, rules are constructed as mutually-recursive functions, and after an alternative rule has been applied, the recognizer backtracks to try another rule. Such recognizers can be constructed as a set of higher order functions, each of which takes an index  $j$  as argument and returns a set of indices. Each index in the result set corresponds to a position at which the parser successfully finished recognizing a sequence of tokens (*input*) that began at position  $j$ . An empty result set indicates that the recognizer has failed. The result for an ambiguous input contains repetition of one or more ending indices.

Using the following basic combinators ( $term_{rec}$  and  $empty_{rec}$  for terminals and empty symbols, and  $\langle | \rangle_{rec}$  and  $\ast \rangle_{rec}$  for alternative rules and sequencing of symbols respectively) as infix operators, recognizers for a subset of CFGs can be constructed as executable specifications:

$$\begin{aligned}
 term_{rec}(t, j) &= \begin{cases} \{\} & , j \geq \#input \\ \{j + 1\}, j^{th} \text{ token of } input = t & \\ \{\} & , \text{otherwise} \end{cases} \\
 empty_{rec} j &= \{j\} \\
 (p \langle | \rangle_{rec} q) j &= (p j) \cup (q j) \\
 (p \ast \rangle_{rec} q) j &= \bigcup (map q (p j))
 \end{aligned}$$

However, recognizers constructed with these basic combinators share the shortcomings of naive top-down parsing: 1) they do not terminate for the left-recursive grammars 2) they require exponential time and space for ambiguous input in the worst case. These problems have been addressed in [3, 4] by use of memoization and a technique that restricts the depth of left-recursion.

### 3 Executable Specifications of General AGs

#### 3.1 Preliminaries

We have extended the work of Frost et al. [4] to declaratively construct modular and executable specifications of fully-general AGs by providing new combinators. Our executable specifications map an input's start position to a set of end positions with tree structures. We also thread attributes (i.e. purely-functional and lazy expressions) along with the start and end positions so that they are available for dependencies that are specified in the semantic rules.

We begin by defining some fundamental data structures. Note that from now on, we use the term *parser* for an executable specification of an attribute grammar rule. At any point in the computation, a parser may have a list of synthesized and inherited attributes *Atts* of any user-defined type. A parser, represented by a label (e.g. *Tree*, *Num* etc.), may have multiple occurrences in a syntax rule, and each occurrence may have different synthesized and/ or inherited attributes. For correct identification, we declare each multiple occurrences as an *Instance*, which is a pair of synthesized/inherited indicator and a unique parser id. For example, an instance of parser *Tree* could be the pair (*Synthesized or Inherited*, *T0*).

All parsers except the root parser may have a list of inherited attributes for a start position *j*, and a list of synthesized attributes associated with each successful end position. To accommodate these attributes, we define data-type *Start* and *End* for a parser by pairing the respective indices with a list of instances and attributes. By definition, a parser produces parse-trees based on syntax rules

to indicate correct derivations. We use a recursive data-type *PTree* that compactly represents parse-trees with each component's attribute values and pointer for *where to go next*. The *Result* of a parser's execution is a mapping from *Start* to a list of *Ends* where each of the *End* results a list of *PTree* structures. A memoization technique (section 4) is used to prevent redundant computations in order to achieve polynomial time for ambiguous input. The memo-table *State* represents a memory space with *Results* for parsers which have succeeded or failed. This table is systematically threaded through parser executions using the standard state-monad [12].

```

data Atts      = MaxVal    {getAVAL :: Int}
               | Binary_OP {getB_OP :: (Int -> Int -> Int)} ...

type InsAttVals = [(Instance, [Atts])]
type Start/End  = (Int, InsAttVals)
type Result     = [(Start, End), [PTree Label]]

data PTree v   = Leaf (v, Instance) | Branch [PTree v]
               | SubNode ((Label, Instance), (Start, End))

```

In the following sections, we describe our approach by defining some higher-order functions with segments of Haskell code. The definitions' syntax is straightforward in nature, and can be followed by using a standard literature on Haskell syntax e.g., [13]. We have defined the functions in a declarative manner so that it would be easier to follow for general audience. The full prototype Haskell implementation can be found at the website mentioned in section 6.

### 3.2 Combinators for Syntax

We use two basic concepts from [4] to accommodate syntax rules including direct and indirect left-recursion :

- To accommodate direct left-recursion, a left-recursive *Context* is used, which keeps track of the number of times a parser has been applied to an input position *j*. For a left-recursive parser, this count is increased on recursive descent, and the parser is curtailed whenever the "*left-recursive count* of parser at *j* exceeds the number of remaining input tokens".
- To accommodate indirect left-recursion, a parser's result is paired with a set of curtailed non-terminals at *j* within its current parse path, which is used to determine the context at which the result has been constructed at *j*.

To maintain the flow of attributes when a parser is re-written by its definition, in addition to being executed on the current *Start* and *Context*, we require that it must pass down its unique id and a list of its own inherited attributes so that they can be used when executing the succeeding parsers' semantic definitions. These inherited attributes are defined in terms of semantic rules when the current parser is part of its predecessor's syntax definition.

The current parser's alternative definitions are formed with the combinator  $\langle | \rangle$ , which not only accommodates alternative syntax rules but also a list of semantic rules associated with each syntax rule. The semantic rules include synthesize rules for the

current parser and inherit rules for parsers in alternative syntax rules. Threading appropriate rules to appropriate parsers is carried out by a combinator called *parser* (section 3.3). Both alternative rules  $p$  and  $q$  are applied to the current position  $j$  and the current context, and the id and inherited attributes of the current parser are also passed down so that they are available to the parsers in both alternatives. All results from  $p$  and  $q$  are merged together at the end. The operation of the combinator  $\langle | \rangle$  can be expressed with the type  $\langle | \rangle :: \text{NTType} \rightarrow \text{NTType} \rightarrow \text{NTType}$ , where :

```

type M a          = Start -> Context -> StateMonad a
type ParseResult = (Context, Result)
type NTType      = Id -> InsAttVals -> M ParseResult

```

In each of the alternative of the current parser, multiple parsers can be sequenced with the sequencing combinator  $*\rangle$ . In the definition of  $*\rangle$  for parsers  $p$  and  $q$ ,  $p$  is first applied to the current start position and the current context. Then  $*\rangle$  enables  $p$  to compute its inherited attributes using a combinator  $nt$  (section 3.3) from an environment of type *SemRule* that consists of  $p$ 's precursor's attributes, and the results of all parsers in sequence with  $p$ . This *Result* contains sequencing parsers' synthesized and inherited attributes that are embedded in *PTree* structures. Because the attributes are treated as lazy and pure expressions,  $p$ 's inherited attribute (or any other parser's synthesized or inherited attribute) computations take place only when they are required somewhere else. The next parser  $q$  is then sequentially applied to the set of end positions returned by  $p$ .  $q$  also computes inherited attributes from the same environment. A result from  $p$  is joined with all subsequent results from  $q$  to form new branch nodes in the tree. The combinator  $*\rangle$ 's input-output relation can be expressed as type  $(*\rangle) :: \text{SeqType} \rightarrow \text{SeqType} \rightarrow \text{SeqType}$ , where :

```

type SemRule = (Instance,(InsAttVals, Id) -> InsAttVals)
type SeqType = Id -> InsAttVals -> [SemRule] -> Result -> M ParseResult

```

The definitions of the AG combinators *term token* and *empty* that define the terminals in the AG rules are analogous to their basic recognizer definitions (section 2.2). The only difference is that the terminals are provided with static synthesized attributes. The *term token* makes sure that these attributes are passed up with the end positions with a tree of type *Leaf*, only if the terminal successfully consumes an input token. In case of *empty*, the synthesized attributes are passed upwards regardless.

### 3.3 Accommodating Arbitrary Dependencies in Semantics

Our syntax-directed evaluation allows semantic rules for a parser to be defined in terms of potentially unevaluated attributes from the current parser, and its predecessor, successors and sibling parsers. We map synthesized and inherited semantic rules associated with parsers in a syntax rule to the starting and ending positions respectively in the parsers' result-sets. Our method of constructing a result for a parser allows us to establish full call-by-need based arbitrary dependencies between attributes - including dependencies from the right and top. For example, when a parser  $p_i$  with a syntax  $p_i = p_m *\rangle p_n$  is applied to position 1 and successfully ends at position 5, one of  $p_i$ 's input/output attribute relations could be :

$$\text{SubNode } p_i (\text{inh}_{p_i 1}, \text{syn}_{p_i 5}) = \text{[..Branch[SubNode } p_m (\text{inh}_{p_m 1}, \text{syn}_{p_m 3}), \text{SubNode } p_n (\text{inh}_{p_n 3}, \text{syn}_{p_n 5})]\text{..]}$$

where, assuming  $p_m$  starts at 1 and ends at 3,  $p_n$  starts at 3 and ends at 5,  $inh_{xy}$  and  $syn_{xy}$  represent inherited and synthesized attributes of parser  $x$  at position  $y$  respectively. From this structure, semantic functions with arbitrary attribute dependencies such as  $inh_{p_m i} \leftarrow f(inh_{p_n j}, syn_{p_i k})$ , (where  $f$  is a desired operation on the attributes) can derive input arguments when required. Note that the output of the example AG from section 1 shows actual result structure. An approach based on strict evaluation, rather than lazy, would not achieve this as it maintains a strict evaluation order.

Each AG rule is formed with a higher order wrapper function *parser*, which primarily maps current parser's synthesized rules to all ending points of the syntax result, and assists each parser in the syntax rule to pass down their inherited rules for future use. A parser's synthesized rules are grouped with the identifier (*Syn, LHS*) from a set of *semantics* that is associated with the current *syntax*. Assuming the syntax would eventually produce a result-set *newRes*, the grouped synthesized rules are mapped to this result using a function *mapSynthesize*. This function computes synthesized attributes by applying the semantic specifications on the succeeding parsers' inherited and/ or synthesized attributes for all *Ptree* entries in the result:

```

parser :: SeqType -> [SemRule] -> Id -> InsAttVals -> M ParseResult
parser syntax semantics id inhAtts j context
= do s <- get
    let ((e,res),s') =
        let sRule = groupRule (Syn, LHS) semantics
            tempRes = syntax id inhAtts semantics res
                ((l,newRes),st) = unState (tempRes j context) s
            groupRule id rules = [rule | (ud,rule) <- rules, id == ud]
        in ((l, mapSynthesize sRule newRes inhAtts id),st)
    put s'
    return (e,res)

```

All parsers in a rule pass down their own identification and a list of inherited attributes so that they can be computed or used in their own definition's semantic rules, if required. This task is done with a higher order function *nt*, which groups the inherited rules for the current parser based on the pair (*Inh, idx*) (where *idx* is the unique *Id* of the parser  $x$ ) from *semantics* of current syntax. Then *nt* facilitates computations of inherited attributes with a mapping function *mapInherited* by applying the grouped rules on a *parser*-provided environment that consists of the predecessor *idp*'s and surrounding parsers' synthesized and inherited attributes. These attributes are to be collected from *newRes*. A parser may have more than one inherited attribute for a particular starting position, which may result from different alternatives. When these attributes are used in any succeeding parser's semantic calculation, they are grouped together under the current parser's single identification so that they are available to carry out desired tasks in the semantic definitions that may require inter-alternative or local result dependencies.

```

nt :: NTType -> Id -> SeqType
nt currentParser idx idp inhAtts semantics newRes
= let inhRules = groupRule (Inh, idx) semantics
    ownInAtts = mapInherited inhRules newRes inhAtts idp
    groupRule id rules = [rule | (ud,rule) <- rules, id == ud]
    in currentParser idx ownInAtts

```

### 3.4 Declarative Executable Specifications of Semantic Rules

We follow a declarative format for the semantic specification which states that synthesized or inherited attribute expressions of a parser can be formed by applying a desired operation on any of the synthesized and/or inherited attributes of any of its surrounding parsers. We define synthesized and inherited semantic expressions with a higher-order function *rule*, which eventually applies user-defined function *userFunction* on lists of attribute values. *rule* is the generalized version of the synthesized and inherited expression constructing combinators *rule\_s* and *rule\_i* respectively, and would ultimately return a value of type `SemRule = (Instance, (InsAttVals, Id) -> InsAttVals)` after attaching appropriate type and id. The argument attributes for *userFunction* are also declaratively specified as synthesized or inherited expressions in *listOfExpr*. These expressions are evaluated with the help of a function *valueOf*, which identifies specified parsers in the user-defined function's argument-expressions either by *LHS* (i.e., when the current parser's attribute is used in the semantics) or by any other parser's unique id in the syntax.

```
rule sORi typ idp userFunction listOfExp
= let formAtts id spec = (id, forNode id . spec)
    forNode id atts = [(id, atts)]
    newVal          = userFunction (map valueOf listOfExp)
  in formAtts (sORi, idp) (setAtt typ. newVal)

valueOf sORi typ id_specified id_current environment
| pIDspec == LHS   = getAttVals (sORi , id_current ) environment typ
| otherwise        = getAttVals (sORi , id_specified) environment typ
```

The user-defined function's argument-expressions are applied to an *environment* of attributes using a recursive function *getAttVals* to collect the specified parsers' respective attributes. As mentioned in the previous section the environment is formed and provided with the help of combinators *parser* and *nt*. The *getAttVals* function collects these attributes by comparing the specified parser's id, synthesized/inherited instance and the desired attribute's type with the similar categories from the environment. These comparison factors are threaded down through the current syntax-directed execution path as unevaluated instructions, and the actual comparison takes place only when the attribute values are requested through user-defined functions.

```
getAttVals :: Instance -> InsAttVals -> (a -> AttValue) -> [AttValue]
getAttVals x ((i,v):ivs) typ =
  let getAtts typ (t:tv) = if (typ undefined) == t
                          then (t :getAtts typ tv)
                          else getAtts typ tv
    getAtts typ []       = []
  in  if (i == x) then getAtts typ v else getAttVals x ivs typ
getAttVals x [] typ     = [ErrorVal "ERROR no id found"]
```

The returned attributes are fed into the operations mentioned in the original semantic rules. These operations are straightforward to define. The only requirement for the construction is that these functions perform the desired task on a list of specifications, which are eventually transformed to a list of attribute values. One example of

these functions could be `findRep`, which converts the specified synthesized maximum value (computed from the predecessor’s alternatives’ result-set) to the current parser’s inherited replacement value:

```
findRep specs = \(atts,i) ->
    RepVal (foldr (max) 0 (map (applyMax atts i) (x:xs)))
applyMax y i x = getAVAL (foldr (getMax)(MaxVal 0) (x y i))
getMax x y = MaxVal (max (getAVAL x) (getAVAL y))
```

Using these combinators and functions, we can now declaratively construct executable language specifications as fully general attribute rules. For instance, all rules for the section 1’s example AG are formed with combinators `*>`, `<|>`, `parser`, `nt` and `rule`. One alternative syntax for  $tree(T_0) ::= tree(T_1) tree(T_2) num(T_3)$  is expressed with `tree = parser (nt tree T1 *> nt tree T2 *> nt num T3)`, and one of the inherited semantics for this syntax  $RepVal.T_1 \downarrow = RepVal.T_0 \downarrow$  is represented with `rule_i RepVal Of T1 Is findRep [inherited RepVal Of LHS]`.

## 4 Use of Memoization

Norvig [14] first showed that Earley-like [15] polynomial time complexity can be achieved in mutually-recursive top-down parsing by using memoization. Frost et al. [16, 3, 4] also employed similar techniques to parser combinators. We utilize a related memoization technique to achieve polynomial time complexity for recursive grammars. We use a state-monad [12] to systematically thread a memo-table of type `[(Label, [(Start, (Context, Result))])]` through all parser executions whilst maintaining pure functionality.

All of our parsers are executed with a wrapper function `memoize`. If the current parser passes the direct left-recursion depth-check test then a `lookup` is performed based on the parser’s `Label` and current position `j` (which resides in `Start`) to retrieve the previously saved `Result`. If there exists a saved result, then that is returned if the indirect left-recursion context-comparison test is satisfied. Otherwise, a new result-set is constructed by applying the parser at `j` with an increased `context` and its own inherited semantics so that they are available for succeeding parsers. The `memoize` function `updates` the memo-table with this new result, inherited semantics and a subset of the current left-rec context corresponding to curtailed non-terminals at the current `j`. The update operation overwrites any previous entry for the current `Label` and `j`, since the current entry would subsume all of the previously computed entries. `memoize` also groups local syntactic ambiguities under `j` in a newly-formed result for a Tomita-like [17] polynomial compact representation, and only returns a reference to this packed entry to the caller, instead of the complete result.

The other task of `memoize` is that, whenever a memoized parser returns a result (either through a lookup or by constructing a new result), it makes sure that the parser’s inherited attributes are integrated with the starting point and the synthesized attributes are accompanied with a correct parser `id` at the ending points in the result-set. When we group the local syntactic ambiguities, we also merge synthesized attributes under the current parser’s identifier.

## 5 Complexity Analysis

Here we informally discuss the worst-case time and space requirements of our algorithm with respect to the length of the input  $n$ . Memoization ensures that a non left-recursive parser is applied to a start position only once. But a left-recursive parser can be applied to the same start position at most  $n$  times due to the depth-check. According to [3], the sequencing combinator  $*>$  performs  $O(n^2)$  operations when applying the second parser to every end position returned by the first parser. Therefore, if there were no semantics involved, then a non left-recursive and a left-recursive parser would require  $O(n^3)$  and  $O(n^4)$  time in the worst-case. While accommodating semantics, we have altered the ambiguity-grouping requirement by collecting distinct attributes resulting in a common end position. This assures the fact the syntactic ambiguity may not necessarily represent semantic ambiguity. In theory, a semantic rule may result in unambiguous attribute values when applied to a group of syntactically ambiguous results, each of whose identical syntactic component may have distinct attribute values. One of the alternative syntax rules  $r ::= p *> q$  may have at most  $n$  syntactic ambiguities, because two parsers' ending positions can be chosen from  $n$  start positions in  $n$  ways. Overall, the number of multiset results for  $r$  is increased from  $n$  to  $n^2$ . The number of ambiguities arising from a single alternative rule with multiple parsers would depend on the number of parsers in sequence, not only on  $n$ . Hence this factor has not been considered in our analysis. If the above parser  $r$  is associated with  $m$  semantic rules, then  $*>$  needs to perform extra  $m * n^2$  operations. Although  $p$  or  $q$ 's all start-end position pairs may be partitioned into multiple multisets, they depend on  $p$  or  $q$ 's syntactic definitions, which are not considered here as operations related to current parser  $r$ . Given a fixed number of semantics, and the highest degree of operation under  $r$  is still  $n^2$ , the time complexities of non left-recursive and left-recursive parsers remain at  $O(n^3)$  and  $O(n^4)$  respectively.

Our *PTree* structure allows us to save results as a list of one-level-depth branches with attribute values attached to pointing sub-nodes. In the memo-table, for each parser's  $n$  input positions, we can store  $n$  branches corresponding to  $n$  end positions. As mentioned earlier, for a branch  $p *> q$ , there are  $n$  possible ambiguities. Hence, we need  $O(n^3)$  space in the worst-case w.r.t. the length of the input. The time and space requirements can be reduced further if we generate only the final semantic value, instead of all possible decorated parse trees, because lazy evaluation would only evaluate those parts of the parse space that are required by the current semantic expression. We suspect that many applications (similar to the one in the next section) would fall under this category.

## 6 Implementation and an Example Application

We have implemented our one-pass top-down AG evaluation algorithm by constructing a set of combinators (as discussed in section 3) in a lazy and purely functional language - Haskell. Using these combinators, declarative specifications can be constructed and executed directly without knowing much about Haskell. To test the usability of our system, we have developed a simplified natural language interface. The syntax of the underlying AG is a fully general CFG that has 15 non-terminals and 32 AG rules, and all syntax rules are associated with a subset of a set-theoretic version of Montague semantics that we have extracted from Frost and Fortier [18]. Our interface is able to answer hundreds of thousands of questions about a particular domain - the *solar*

*system*. More information about the implementation and this application, and a version of demo code can be found at <http://cs.uwindsor.ca/~hafiz/fullAg.html>.

We define an attribute type as a set of alternative attributes, where each has its own function type. These attributes are the type-definitions of semantic expressions, which propagate up or down during parser executions. For example:

```
data Att = TERMPHJOIN_VAL {getTJVAL :: ((ES -> Bool) ->
                                   (ES -> Bool) -> (ES -> Bool))}
        | QUEST_VAL      {getQUVAL :: String}....
```

Next we construct a dictionary to define syntactic categories and their meanings e.g.,

```
dictionary = [("man",  Cnoun, [NOUNCLA_VAL set_of_men])
             ,("orbit", Tverb, [VERB_VAL (tran_verb rel_orbit)]),
             ,("human", Cnoun, meaning_of_nouncla "man or woman" Nouncla)
             ,...]
```

Then we modularly define a complete AG specification for the solar system application. For example, part of the definitions of term-phrase and noun-clause are:

```
jointermph =
memoize Jointermph
parser (nt jointermph S1 *> nt termphjoin S2 *> nt jointermph S3)
[rule_s TERMPH_VAL Of LHS Is
 appjoin1 [synthesized TERMPH_VAL Of S1
           , synthesized TERMPHJOIN_VAL Of S2
           , synthesized TERMPH_VAL Of S3]]
<|>
parser (nt termph S4)
[rule_s TERMPH_VAL Of LHS Is
 copy [synthesized TERMPH_VAL Of S4]]

snouncla =
memoize Snouncla
parser (nt adjs S1 *> nt cnoun S2)
[rule_s NOUNCLA_VAL Of LHS Is
 intrsct1 [synthesized ADJ_VAL Of S1
           , synthesized NOUNCLA_VAL Of S2]]
<|> ...
```

Being right and left recursive, the parser `jointermph` expands to both right and left. The semantic expressions are declaratively defined e.g., `jointermph`'s first semantic rule expresses that `jointermph`'s synthesised attribute `TERMPH_VAL` is formed by joining the synthesized attributes of the r.h.s parsers `S1`, `S2` and `S3`. The operations, which are applied to syntactic symbols' attributes, are defined based on a set-theoretic version of Montague semantics. For example, `snouncla`'s attribute `NOUNCLA_VAL` is obtained by intersecting sets of adjectives and common-nouns.

An example session with our interface is as follows:

```

which moons that were discovered by hall orbit mars => [phobos deimos]
every planet is orbited by a moon                  => [false]
how many moons were discovered by hall or kuiper    => [4]
did hall discover deimos or phobos and miranda      => [no, yes]
etc.

```

Note that the last answer is ambiguous due to the right and left branching of `jointermph`, hence are separated by a comma.

## 7 Related Work

The primary use of AGs has mostly been the specification and construction of compatible parser-generators for programming languages [19]. The classical definition of an AG has often been modified to support the needs of such languages. Swierstra et al. introduced the idea of *higher order attributes* [20, 21] by treating syntax as a part of semantic functions' input and output in a semantics-driven analysis. De Moor et al. [6] achieved semantic modularity by treating attributes as first-class objects. Boyland [22], described an efficient method - *collections* for remote attribute dependencies. *JustAdd* [23] is a compiler-compiler AG system for Java that supports circular referential dependencies with conditional rewriting of ASTs using lazy-evaluation. The *Silver* specification language [24] has been developed primarily based on *forwarding* (a concept similar to higher-order attributes) and other extensions mentioned above. Kats et al. [25] describe *attribute decorators* that support many AG extensions. Similar to *JustAdd*, they use memoization for efficient attribute evaluation.

Our approach differs from these approaches by offering a platform that strictly preserves the syntactic structure of ambiguous CFGs (which includes direct and in-direct left-recursions). Our top-down syntax-driven parsing strategy provides a set of non-strict combinators for constructing fully declarative semantic expressions with arbitrary dependencies. In addition to eliminating redundant computations, our use of memoization technique has been specialized to perform extra tasks such as keeping track of non-terminals' context information, merging syntactic ambiguity, mapping and grouping attributes etc.

Even though use of lazy-evaluation to build AG systems has been around for a long time [26, 27, etc.], little work has been done using AGs for natural language processing tasks: Levison and Lessard [28] used AGs to impose some degree of grammatical and semantic agreement by propagating only inherited attributes downwards while generating natural language text. In the template-based natural language generating system *YAG* [29], AGs have been used to correct partially-specified input by imposing grammatical/number restrictions [30]. Their multi-pass evaluating process begins by initializing inherited attributes with values from the input, then evaluating the rest of the input.

Our approach differs from the last two approaches by being a complete one-pass parsing system that can return either compactly-represented parse trees with attribute values in nodes or just the final answer(s). This is in contrast to the template-based text generators which receive structured input, not natural languages sentences, and don't use AGs for full-blown parsing. By being *lazy*, we achieve general attribute dependencies by providing more flexible input/output attribute relations. Also, along with declarative semantics, our syntax is highly modular because of the systematic use of parser-combinators as basic building blocks.

## 8 Concluding Comments

We have developed a framework where general CFGs (including ambiguous and left-recursive grammars) can be integrated with semantic rules with arbitrary attribute dependencies as directly-executable and modular specifications. Our approach is based on a top-down parsing method implemented as a set of non-strict combinators resulting in declarative specifications. We utilize a memoization technique for polynomial time and space complexities. In the future we aim to process syntactic and semantic ambiguities based on grammatical and number agreement, type checking and conditional restrictions. By taking advantage of arbitrary attribute dependencies, we plan to model NL features that can be characterized by other grammar formalisms such as unification grammars, combinatory-categorical grammars and type-theoretic grammars. We are constructing formal correctness proofs, and optimizing the implementation for using with very large grammars. We believe that our work will help computational linguists build and test their theories and specifications without worrying about the underlying computational methods, and will also help non-experts create NL interfaces to their applications.

## 9 Acknowledgements

The authors would like to thank the referees for their constructive criticisms and helpful suggestions. Richard Frost and Rahmatullah Hafiz thank the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Government of Ontario, respectively, for their support.

## References

1. Knuth, D.: Semantics of context-free languages. *Theory of Computing Systems*, Springer New York **2**(2) (1968) 127–145
2. Warren, D.: Programming the ptq grammar in xsb. In: *Workshop on Programming with Logic Databases*. (1993) 217–234
3. Frost, R., Hafiz, R., Callaghan, P.: Modular and efficient top-down parsing for ambiguous left-recursive grammars. *10th IWPT, ACL*. (2007) 109 – 120
4. Frost, R., Hafiz, R., Callaghan, P.: Parser combinators for ambiguous left-recursive grammars. *10th PADL, ACM*. **4902** (2008) 167–181
5. Bird, R.: *Intro. to Functional Programming using Haskell*. Prentice Hall (1998)
6. De Moor, O., Backhouse, K., Swierstra, D.: First-class attribute grammars. In: *Third Workshop on Attribute Grammars and their Applications*. (2000) 245–256
7. Tienari, M.: On the definition of attribute grammar. *Semantics-Directed Compiler Generation* **94** (1980) 408 – 414
8. Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher order attribute grammars. In: *PLDI, ACM* (1989) 131–145
9. Frost, R., Launchbury, J.: Constructing natural language interpreters in a lazy functional language. *The Computer Journal* **32**(2) (1989) 108–12
10. Hutton, G., Meijer, E.: Monadic parser combinators. *J. Funct. Program.* **8**(4) (1998) 437 – 444
11. Wadler, P.: How to replace failure by a list of successes. In: *Functional programming languages and computer architecture*. Volume LNCS 201. (1985) 113 – 128

12. Wadler, P.: Monads for functional programming. First International Spring School on Advanced Functional Programming Techniques **925** (1995) 24 – 52
13. Hudak, P., Peterson, J., Fasel, J.: A gentle introduction to haskell 98. Technical report (1999)
14. Norvig, P.: Techniques for automatic memoization with applications to context-free parsing. Computational Linguistics **17(1)** (1991) 91 – 98
15. Earley, J.: An efficient context-free parsing algorithm. Commun. ACM **13(2)** (1970) 94–102
16. Frost, R., Szydlowski, B.: Memoizing purely functional top-down backtracking language processors. Science of Computer Programming **27(3)** (1996) 263–288
17. Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Boston, MA. (1986)
18. Frost, R., Fortier, R.: An efficient denotational semantics for natural language database queries. In: Applications of NLDB. (2007) 12–24
19. Paakki, J.: Attribute grammar paradigms a high-level methodology in language implementation. ACM Comput. Survey **27(2)** (1995) 196–255
20. Swierstra, S.D., Alcocer, P., Saraiva, J.: Designing and implementing combinator languages. In: 3rd Summer School on Advanced FP. (1998) 150 – 206
21. Swierstra, S.D., Vogt, H.: Higher order attribute grammars. In: Attribute Grammars, Applications and Systems. Volume 545. (1991) 256–296
22. Boyland, J.: Remote attribute grammars. Journal of the ACM **52(4)** (2005) 627 – 687
23. Ekman, T.: Extensible Compiler Construction. PhD thesis, Comp Science, Lund University (2006)
24. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an extensible attribute grammar system. In: LDTA. (2007) 103 – 116
25. Kats, L., Sloane, A., Visser, E.: Decorated attribute grammars. attribute evaluation meets strategic programming. In: 18th International Conference on Compiler Construction. (2009) 142 – 157
26. Augusteijn, L.: The elegant compiler generator system. In: Attribute Grammars and their Applications. (1990) 238–254
27. Johnsson, T.: Attribute grammars as a functional programming paradigm. In: FP languages and computer architecture. Volume 274. (1987) 154 – 173
28. Levison, M., Lessard, G.: Application of attribute grammars to natural language sentence generation. AGs and their Applications **461** (1990) 298–312
29. Mcroy, S., Channarukul, S., Ali, S.: An augmented template-based approach to text realization. Natural Language Engineering. **9(4)** (2003) 381 – 420
30. Channarukul, S., Mcroy, S., Ali, S.: Enriching partially-specified representations for text realization using an attribute grammar. In: 1st International Natural Language Generation Conference. (2000) 163 – 170