

# A System for Modularly Constructing Efficient Natural Language Processors

Rahmatullah Hafiz  
School of Computer Science  
University of Windsor  
Windsor, ON, Canada  
Email: hafiz@uwindsor.ca

Richard A. Frost  
School of Computer Science  
University of Windsor  
Windsor, ON, Canada  
Email: richard@uwindsor.ca

**Abstract**—A system based on a general top-down parsing algorithm has been developed which allows language processors to be created as executable specifications of arbitrary attribute grammars. Attribute grammars allow the declarative definition of languages. Syntax is defined through context-free grammar rules, and meaning is defined by associated semantic rules. The executable specifications are highly modular. Innovative techniques enable the efficient accommodation of left-recursive rules, ambiguity, and arbitrary semantic dependencies. A new technique allows parses to be pruned by arbitrary semantic constraints. This new technique is useful in modelling natural-language phenomena by imposing unification-like restrictions, and accommodating long-distance and cross-serial dependencies, which cannot be handled by context-free rules alone.

**Index Terms**—Top-down parsing; Attribute grammars; Lazy evaluation; Constraint-based Formalism; Compositional Semantics

## I. INTRODUCTION

Traditionally top-down processing has been used for generative grammars (e.g., definite clause grammars (DCGs)[1], and context free grammars (CFGs)) to model natural languages. From a computational point of view, top-down analysis of languages showed early potential by being modular, and by accommodating a wide range of linguistic phenomenon that are essential for deep analysis. Moreover, modular top-down analysis allows construction of larger language processors by piece-wise combination of smaller components. This type of construction is especially useful when processors (e.g., natural language interfaces) compute meanings of sentences using compositional semantics such as Montagovian semantics. Specifying syntax and semantics to describe formal languages using denotational notation of attribute grammars (AGs) has been practiced extensively. However, very little work has shown the usefulness of declarative AGs for computational models for natural language. Previous work falls short in accommodating ambiguous CFGs with left-recursive rules, and providing a declarative syntax-semantics interface that can take full advantages of dependencies between syntactic constituents to model linguistically-motivated cases.

In this paper we show that a unique top-down parsing approach can be used to build a system<sup>1</sup> in a purely functional language Haskell [2] where application developers can specify

syntactic and semantic descriptions of natural languages using a general notation of AGs as directly executable specifications. The underlying top-down analysis method parses ambiguous sentences with general CFGs efficiently and allows coupling semantic rules with syntax declaratively. This system can be used not only to produce compact representation of the potentially exponential number of parses using polynomial time and space, and to compute meanings of sentences using compositional semantics, but also to model linguistic properties that may require more restricted generative formalisms than context-free grammars. Some of these properties we demonstrate by only using upward propagating synthesized attributes include characteristics of unification-based formalisms (e.g., subject-verb agreements), unbounded syntactic dependencies in sentences for some form of disambiguation, and generative formalisms that can process cross-serial dependencies and duplicate languages. We also emphasize how executable language processors can be constructed by integrating compositional semantics with a context-free backbone.

Our underlying technique relies on 1) constructing a set of higher-order pure functions that allow the construction of modular and directly-executable specifications of syntax and semantics rules, 2) a context and depth-aware top-down parsing algorithm for general CFGs, and 3) a variation of explicit memorization for time and space efficiency. The lazy-evaluation procedure is the necessary technique for us to impose constraints to model targeted languages, which implicitly makes sure that computations are carried away out when they are asked for (call-by-need) and, a particular computation is performed at most once and its result is recalled whenever it is needed again. Our declarative notation allows designers to specify *what* the description of the language is, *what* relationships exist between components, and *what* restrictions are needed to be imposed, rather than *how* these procedures actually work.

## II. GENERAL NOTATION

Here we introduce the notation that we will use in our system to represent directly-executable specifications of AGs.

<sup>1</sup>Visit [cs.uwindsor.ca/~hafiz/xsaiga/fullAg.html](http://cs.uwindsor.ca/~hafiz/xsaiga/fullAg.html) for the system code.

## A. Operators for Syntactic Analysis

Consider the following context-free grammar (in BNF) for a segment of English that contains sentences such as bob saw a nightingale:

```

sent ::= tp vp
tp   ::= pnoun |det np
vp   ::= verb tp
np   ::= noun
pnoun ::= 'bob'
noun  ::= 'nightingale'
verb  ::= 'saw'
det   ::= 'a'

```

Example CFG 1

In the above grammar, a nonterminal sentence (*sent*) is a term phrase(*tp*) followed by a verb phrase (*vp*). A *tp* can be expanded using two alternative rules - one rewrites *tp* to a proper noun *pnoun*, and in the other one *tp* is formed by a determiner (*det*) followed by a noun phrase (*np*). The alternatives are separated with a | and terminals are in single quot. When *sent* is applied to bob saw a nightingale, it will unambiguously produce a single parse tree.

In our system for executable specifications, we provide a set of higher order functions or combinators to modularly denote syntactic descriptions similar to the above example. We use the combinators <|> and \*> to denote alternating and sequencing of syntax symbols, and term to represent a terminal. We will gradually increase this set of combinators or operators to accommodate other linguistic properties. Using these basic combinators, we can represent CFG 1 as the following executable specification:

```

sent = memoize Sent (tp *> vp)
tp   = memoize Tp   (pnoun <|> det *> np)
vp   = memoize Vp   (verb*> tp)
np   = memoize Np   noun
pnoun = term "bob"
noun  = term "nightingale"
verb  = term "saw"
det   = term "a"

```

In this executable specification, each expression is a function for a non-terminal that maps an index representing the current input position (*Start*) to a parse-tree of the recursive data-type *Ptree* (see below) that indicates the structure of the current parse that successfully ends at the position *End*. The final result of complete parsing is a list of these *Ptrees* with their starting and ending positions. The embedded (*Start, End*) pairs in trees work as pointers to indicate where to go next, which allows the trees in the result set to be one-level-depth branches, sub-nodes or leaves. The trees that start and end at identical positions are shared between all non-terminal entries. Note that each non-terminal's functional definition is *memoized* with a wrapper function *memoize* that ensures that a non-terminal (identified by a unique label e.g., *Sent*) is executed at a input position at most once.

```

type Start/End = Int
data PTree Label = Leaf Label
                  | Branch[PTree Label]
                  | SubNode(Label, (Start, End))
type Result     = [(Start, End), [PTree Label]]

```

The compact and memoized result for all successful parses is systematically threaded through parser-executions. As pure functional combinators do not have side effects, the threading of the memoized table is done with a *state monad* [3].

We can extend CFG 1 by adding rules for prepositional phrases *pp* so that we can parse a wide range of sentences such as bob saw a nightingale with a telescope. We can also add more terminals according to the target language that we want to model:

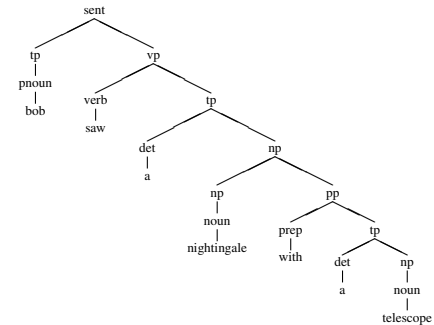
```

sent ::= tp vp
tp   ::= pnoun |det np
pp   ::= prep tp
vp   ::= vp pp |verb tp
np   ::= np pp |noun
pnoun ::= 'bob'
noun  ::= 'nightingale' |'telescope'
prep  ::= 'with' |'on' |etc.
verb  ::= 'saw' |'see' |etc.
det   ::= 'the' |'an' |'a'

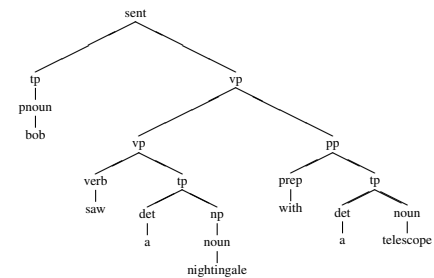
```

Example CFG 2

This extended CFG 2 now contains left-recursive rules (e.g., *vp::=vp pp*), and it can accommodate ambiguity. For example, when we parse the sentence bob saw a nightingale with a telescope starting from the non-terminal *sent* using a general parsing algorithm we get the following two parses:



Parse Tree 1.



Parse Tree 2.

As our system allows piecewise integration of smaller components to form a larger description, we can seamlessly extend our previous executable specification of syntax to model CFG 2 (as shown below). This form of modular construction also allows testing and editing of individual components separately without affecting other parts.

```

sent = memoize Sent (tp *> vp)
tp   = memoize TP   (pnoun <|> det *> np)
pp   = memoize PP   (prep *> tp)

```

```

vp = memoize VP (verb *> tp <|> vp *> pp)
np = memoize NP (noun <|> np *> pp)
pnoun = memoize "pnoun" term "bob"
noun = memoize "noun" term "nightingale" <|> term "telescope"
prep = memoize "prep" term "with" <|> etc.
verb = memoize "verb" term "saw" <|> etc.
det = memoize "det" term "a" <|> etc.

```

This specification can be executed directly using our general top-down parsing algorithm ([4]) to accommodate a subset of English that contains ambiguity. When the starting nonterminal `sent` is applied at the first token position of the sentence `bob saw a nightingale with a telescope`, it produces two parses that are embedded in the following pretty-printed compact format. Two different parses can be extracted if we follow the  $(Start, End)$  pointing pairs starting from the `Sent`'s entry. Notice that ambiguities arise from two of the `vp`'s entries where the verb phrase starts at position 2 and ends at position 8 for the sub-string `saw a nightingale with a telescope` using two different sub-trees. Each entry in this compact representation is shared i.e., they are entered exactly once in the result set. For example, the prepositional phrase `pp` that starts at position 5 and ends at position 8, is shared by the `vp` and the `np`.

```

"Sent" (Start: 1,End: 8) Branch [SubNode ("tp", (1,2))
                               ,SubNode ("vp", (2,8))]
"VP"   (Start: 2,End: 8) [Branch [SubNode ("verb", (2,3))
                               ,SubNode ("tp", (3,8))]
                             ,Branch [SubNode ("vp", (2,5))
                                       ,SubNode ("pp", (5,8))] ] ]
      (Start: 2,End: 5) Branch [SubNode ("verb", (2,3))
                               ,SubNode ("tp", (3,5))]
"TP"   (Start: 1,End: 2) SubNode ("pnoun", (1,2))
      (Start: 3,End: 8) Branch [SubNode ("det", (3,4))
                               ,SubNode ("np", (4,8))]
      (Start: 3,End: 5) Branch [SubNode ("det", (3,4))
                               ,SubNode ("np", (4,5))]
      (Start: 6,End: 8) Branch [SubNode ("det", (6,7))
                               ,SubNode ("np", (7,8))]
"NP"   (Start: 4,End: 8) Branch [SubNode ("np", (4,5))
                               ,SubNode ("pp", (5,8))]
"PP"   (Start: 5,End: 8) Branch [SubNode ("np", (5,6))
                               ,SubNode ("pp", (6,8))]
"pnoun" (Start: 1,End: 2) Leaf "bob"
"verb"  (Start: 2,End: 3) Leaf "saw"
"det"   (Start: 3,End: 4) Leaf "a"
"noun"  (Start: 4,End: 5) Leaf "nightingale"
"prep"  (Start: 5,End: 6) Leaf "with"
"det"   (Start: 6,End: 7) Leaf "a"
"noun"  (Start: 8,End: 9) Leaf "telescope"

```

## B. Operators for Semantic Analysis

1) *Computing Meaning*: Our system is ideally suited to incorporate compositional semantics that have a one-to-one correspondence between the rules defining syntactic constructs and the rules stating how the meaning of phrases are constructed from the meanings of their constituents. Richard Montague, who was one of the first to develop a compositional semantics for English, suggested to treat natural language semantics similar to formal language semantics as purely-functional  $\lambda$  expressions where larger expressions are constructed by composing smaller expressions. As the original

proposal is computationally intractable, we use an efficient set-theoretic version [5].

Alongside with piecewise syntactic extension, we can glue together semantics rules for corresponding syntax rules to compute meanings of a larger set of languages. We extend the set of combinators to integrate semantics with syntax as directly-executable specifications of general attribute grammars. Consider a subset of CFG  $2$   $sent ::= tp\ vp$ ,  $vp ::= verb\ tp$ ,  $tp ::= pnoun\ det\ noun$  that can parse the sentence `bob saw a nightingale`. Following the suggestions of [5], we define an AG where syntax rules are accompanied by sets of semantic rules, which are Montague-style compositional semantics as  $(\lambda)$  expressions:

```

sent(S0) ::= tp(T0) vp(V0)
           {S0.VAL ↑ = (λp p T0.VAL ↑) V0.VAL ↑}
tp(T0)   ::= pnoun(P0)
           {V0.VAL ↑ = P0.VAL ↑}
           | det(D0) noun(N0)
           {T0.VAL ↑ = (λp D0.VAL ↑ p) N0.VAL ↑}
vp(V0)   ::= verb(V1) tp(T1)
           {V0.VAL ↑ = (λp V1.VAL ↑ p) T1.VAL ↑}
pnoun(P0) ::= 'bob'
           {P0.VAL ↑ = (λp 'bob' ∈ p) {human}}
noun(N0)  ::= 'nightingale'
           {V0.VAL ↑ = {nightingales}}
verb(V0)  ::= 'saw'
           {V0.VAL ↑ = (λz z(λxλy applytransvp(y,x)))}
det(D0)   ::= 'a'
           {D0.VAL ↑ = λpλq(p ∧ q) ≠ φ}

```

### Attribute Grammar 1.

In the above AG, non-terminals are identified with labels (e.g.  $V_0$ ,  $N_0$  etc.), and these non-terminals have synthesized attributes (e.g.  $V_0.VAL \uparrow$ ) that propagate upward. If inherited attributes are needed that propagate downward then we can mark them with  $\downarrow$ s. The semantics for non-terminals are correctly-typed pure functions, and some of them are higher-order functions. For example, the semantic rule for the transitive verb `saw`, which is identified with the  $V_0.VAL$ , is a function that receives two functions - one defines the term phrase `a nightingale` and the other defines the proper noun `bob` as input arguments. The semantics for `saw` eventually passes the input arguments into a user-defined function `applytransvp` that determines the authenticity of the transitive verb's relation between the subject and the object.

We extended our set of combinators to accommodate AGs like the above, which are described in section 4. We have built combinators `rule_s` and `rule_i` to form synthesized and inherited semantic rules, wrapper functions `parser` and `nt` to form a complete AG expressions, and we refer to syntactic constructs by their unique labels (e.g.,  $V_1$ ,  $P_1$  etc.) in order to use them as semantic functions' arguments. By allowing any syntactic construct's attributes that are available in a syntax rule as a semantic functions' arguments, we can form arbitrary dependencies between these constructs that are necessary for many linguistic phenomena. The above AG can be described declaratively with our notation in Haskell's syntax to form a directly-executable specification. Below we show only a

few AG rules (the rules for nonterminals *sent*, *tp*, and *vp*) to introduce our notation for integrating semantic rules with syntax rules:

```

1. sent = memoize Sent
2. (parser (nt tp T0 *> nt vp V0)
3. [rule_s val OF LHS EQ applytp
4. [synthesized VAL OF V0, synthesized VAL OF T0]]
5. tp = memoize TP
6. (parser (nt det D0 *> nt noun N0)
7. [rule_s val OF LHS EQ applydet
8. [synthesized VAL OF D0, synthesized VAL OF N0]]
8. <|>
9. parser (nt pnoun P0)
10. [rule_s Ref OF LHS EQ copy
11. [synthesized Ref OF P0]]]
12. vp = memoize VP
13. (parser (nt verb V1 *> nt tp T1)
14. [rule_s val OF LHS EQ applyvp
15. [synthesized VAL OF V1, synthesized VAL OF T1]]
.....

```

In the above specification, lines 1, 2, 5, 6, 9, 12, and 13 are syntax rules for *sent*, *tp*, and *vp*, and the rest are sets of associated declarative semantics. For example, the synthesized VAL attributes of non-terminals (which are identified by LHSs) are computed with user functions (such as `applytp`) by supplying the required syntactic constructs’ attributes (such as `synthesized VAL OF V0`). These user functions perform similar tasks that are mentioned as  $\lambda$  expressions in Attribute Grammar 1. We have constructed sample NL interfaces using our declarative AG notation that uses context-free syntax and compositional semantics. These applications can answer hundreds of thousands of questions about particular domains (see the web-link at footnote 1 for a sample application implementation).

2) *Using Semantics for Disambiguation:* Here we provide a simple example to show an innovative use of our declarative semantic notation to perform one form of natural language disambiguation.

The sentence *bob saw a nightingale with a telescope* that we parsed using CFG 2, which produced two syntax trees, can be interpreted in more than one way. In a prepositional phrase *pp*, a preposition demonstrates an adjectival or adverbial property by quantifying either the noun phrase *np* or the verb phrase *vp*. According to the first parse, the *np*’s head noun *nightingale* is quantified by the preposition *with*, which can be misinterpreted as “a telescope carrying nightingale”. But in the second parse, the *with* quantifies the verb *saw* from the *vp*, which is semantically more meaningful. Based on this argument, we can make sure that the first parse is discarded while generating only the second parse by using the following attribute grammar:

```

sent(S0) ::= tp(T0) vp(V0)
tp(T0) ::= det(D0) np(N0) | pnoun(N1)
pp(P0) ::= prep(P1) tp(T0) {P0.Ref ↑ = T0.Ref ↑}
vp(V0) ::= vp(V1) pp(P0)
{V0.Ref ↑ = V1.Ref ↑
 ,V0.Kill ↑ = V1.Ref ↑ ∉ P0.Ref ↑}
{V0.Ref ↑ = V1.Ref ↑}
np(N0) ::= np(N1) pp(P0)
{N0.Ref ↑ = N1.Ref ↑
 ,N0.Kill ↑ = N1.Ref ↑ ∉ P0.Ref ↑}
|noun(N1)

```

```

{N0.Ref ↑ = N1.Ref ↑}
pnoun(N0) ::= bob
noun(N0) ::= nightingale {N0.Ref ↑ = bird}
|telescope {N0.Ref ↑ = used to see}
verb(V0) ::= saw {N0.Ref ↑ = to see}
prep(V0) ::= with
det(D0) ::= a

```

## Attribute Grammar 2.

In the above AG, we added semantic rules for agreements with the verb phrase *vp* and the noun phrase *np* that result in rejection of the sub-parse that produces an attribute *Kill* of the value *true*. According to our approach, when a sub-parse is rejected, then the entire parse is rejected too. These rejection rules check to see whether prepositional phrase *pp* is meaningfully quantifying the neighboring verb phrase and noun phrase by matching their upward propagating *Ref* attribute values. In this artificial example, we wanted to enforce the requirement that the noun *telescope* has a meaningful match with the verb *saw* (i.e., the *saw*’s attribute belongs to the *telescope*’s attribute), but not with the noun *nightingale*. In subsequent sections we describe the working mechanism of our approach. The above AG can be described modularly with our notation to create the following declarative specification (we only show the rule for verb phrase *vp*):

```

vp = memoize VP
(parser (nt vp V1 *> nt pp P1)
[rule_s Ref OF LHS EQ copy [synthesized Ref OF V1]
 ,rule_s Kill of LHS EQ notElemOf
 [synthesized Ref OF, synthesized Ref OF P1]]
<|>
parser (nt verb V1 *> nt tp T1)
[rule_s Ref OF LHS EQ copy [synthesized Ref OF V1]
 ,rule_s Kill of LHS EQ False ])

```

Here the second semantic rule for *vp* can discard a parse by checking whether the *Ref OF V1* is a member of *Ref OF P1* with the function *notElemOf*. When this *vp* rule is placed with other rules to form the complete executable AG, and when the root non-terminal *sent* is applied to our example sentence, then it will only generate one compactly-represented parse tree similar to the “Parse Tree 2” on the previous page.

## III. BACKGROUND

In this section we briefly describe some related grammar formalisms [6] and the general top-down parsing technique.

### A. Grammar Formalisms

Our approach is based on a context-free backbone. That means the targeted natural language can be generated, recognized, parsed and computed using context-free generative syntax rules. A context-free grammar (CFG)  $G$  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  $b \in (N \cup T)^*$ .

In an attribute grammar (AG, [7]), syntax rules of CFGs are augmented with semantic rules to describe the meaning of the sentences of a context-free language. Although different

definitions are given in [8], [9], [10], we prefer to define a general AG by imposing minimal restrictions on attribute dependencies. An AG is a 3-tuple  $AG = (G, A, R)$ , where  $G$  is the founding CFG,  $A$  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/expression  $r \in R$ . The set  $A(X)$  can be partitioned into two sets  $A_i(X)$  and  $A_s(X)$ , which represent *inherited* and *synthesized* attributes respectively. A synthesized attribute is an attribute for the LHS non-terminal of a production rule, and it can be computed using any attributes from the RHS terminals/non-terminals. Whereas an inherited attribute is associated with a terminal/non-terminal that resides at the RHS of the production rule, which can be computed attributes of terminals/non-terminals that are on the right or on the left of the current non-terminal. In conventional term, inherited attributes propagate downwards and synthesized attributes propagate upwards.

From the viewpoint of pure functions and lazy-evaluation, semantics rules and attributes are interchangeable. When we refer to an attribute we either refer to a value - *static semantics*, or to an unevaluated expression - *dynamic semantics*.

### B. Top-Down Parsing with Parser Combinators

Parser combinators have been used extensively ([11], [12], [13] etc.) to prototype top-down backtracking functional recognizers and parsers to provide modular and piecewise construction of executable specifications of grammars that can accommodate ambiguity. In basic recursive-descent recognition, syntax rules are constructed as mutually-recursive functions, each of which maps an starting *input* position to a set of indices corresponding to a set of ending positions at which the parser successfully finished recognizing a sequence of *input* tokens. An empty result set indicates that the recognizer has failed. After an alternative rule has been applied, the recognizer backtracks to try another rule. The result for an ambiguous input contains one or more ending indices. In the case of parsing, indices in the result set are replaced by tree structures to show successful parsing structures.

However, top-down recognizers constructed with basic combinators do not terminate for left-recursive grammars, and when extended to parsers they require exponential time and space for ambiguous input in the worst case. We have addressed these problems [14] by the use of a technique that restricts the depth of left-recursive calls, curtails a parse when left-recursive call exceeds the number of remaining input tokens, and that tracks curtailed indirect left-recursive non-terminals to determine the context at which the result has been constructed. We also use memoization techniques for parsers (a technique similar to [15]) that ensures  $O(n^3)$  and  $O(n^4)$  worst-case time complexities for non left-recursive and left-recursive CFGs respectively. For space efficiency, the potentially exponential number of ambiguous results are represented compactly in a directed acyclic graph with polynomial size space where entries are one-level-depth branches or sub-nodes, and ambiguous entries are grouped under common nodes.

These nodes have referencing pointers to construct complete parse trees whenever needed (similar to [16]).

## IV. DECLARATIVE AND EXECUTABLE ATTRIBUTE GRAMMARS

### A. Forming Arbitrary Dependency

In [17], we demonstrated how declarative semantics rules can be integrated with CFGs' syntax rules in top-down parsing by describing the implementation of combinators in the purely functional language Haskell that enable the formation of language specifications with modular syntax and semantic. The new parser<sup>2</sup> now has a set of attributes, which are computed from other terminals/non-terminals' attributes that belong to the current parser's syntax definition(s).

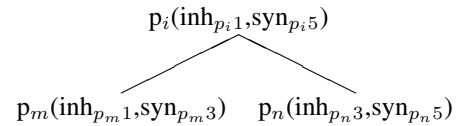
Even if a parser contains semantic rules as well as semantic rules, the result of a parser's execution is still a mapping from a start position of input to a set of parse-tree structures with end points. A recursive parse-tree can be a single leaf (to represent a terminal), a sub-node (to represent a non-terminal), or a branch (to represent combinations of terminals and non-terminals, which models more complex syntax). The primary change in the tree data structure is how attributes are represented and threaded so that they are available for dependencies that are specified in the semantic rules. These attributes are essentially purely-functional lazy expressions. For example, the `BinaryOP` attribute in the example below is a function that computes an integer by applying a binary operation to two input integers.

```

type Start/End = (Int,[Attributes])
data Attributes = MaxVal {getAVAL :: Int}
                | BinaryOP {getBOP :: (Int → Int → Int)}

```

Synthesized and inherited semantic rules associated with a parser are mapped onto the starting and ending positions respectively in the parser's result-set. This facilitates the overall syntax-directed evaluation and allows semantic rules to be denoted in terms of potentially unevaluated attributes from the environment of the current parser, its predecessor, successors, and sibling parsers. For example, when a parser  $p_i$  with a syntax rule  $p_i = p_m * > p_n$  is applied to position 1 and successfully ends at position 5, one of  $p_i$ 's input/output attribute relations might be:



where, assuming  $p_m$  starts parsing at 1 and ends at 3,  $p_n$  starts parsing at 3 and ends at 5,  $\text{inh}_{xy}$  and  $\text{syn}_{xy}$  represent inherited and synthesized attributes of parser  $x$  at position  $y$  respectively. From this structure, semantic functions with arbitrary attribute dependencies (like the one below) can acquire input arguments whenever required while they are integrated with the syntax rule:

```

p_i      = p_m * > p_n
inh_{p_m i} ← function(inh_{p_n j}, syn_{p_i k})

```

<sup>2</sup>From now on we shall refer non-terminals of attribute grammars as parsers.

## B. Construction of Combinators for Executable AGs

Here we informally discuss the construction of the combinators. The alternative combinator  $\langle | \rangle$ 's inputs  $p$  and  $q$  are alternative syntax with lists of respective semantic rules.  $p$  and  $q$  are executed with the current *start* position and a *context* (for left-recursive calculation), and they pass down their ids and inherited rules so that they can be used in a succeeding parsers' semantics. The associated semantic rules include synthesized rules for the parent parser (referred as *LHS*) and inherited rules for parsers that are in the associate syntax rule. Threading appropriate semantic rules to the appropriate syntax symbol is done with the combinator *parser*. All results from alternative parsers are merged together at the end.

Terminals or non-terminals (say  $p$  and  $q$ ) can be sequenced with the combinator  $\ast >$  to form compound a syntax. This combinator enables  $p$  to be applied to the current *start* position and *context*, and to compute its inherited attributes non-strictly using the combinator *nt* from an environment that consists of  $p$ 's precursor's attributes, and the *result* of the current syntax. This *result* (a tree structure) contains all sequencing parsers' synthesized and inherited attributes. The next parser  $q$  is then sequentially applied to the set of end positions returned by  $p$ , and computes inherited attributes from the same environment using the combinator *nt*. A result from  $p$  is united with all subsequent results from  $q$  to form new branches in the tree.

The higher-order function *nt* forms a non-terminal of an AG that enables parsers to pass down their own identification and a list of inherited semantic rules so that they can be used in subsequent AG definitions. These rules are evaluated by applying them on an environment that consists of the predecessor's and surrounding parsers' attributes. This function ultimately maps the current parser's inherited attributes to the starting point of a result so that other inquiring parsers know where to look if needed in the future. The wrapper function *parser* forms a complete AG rule by mapping the current parser's synthesized rules to the ending points of the result, and by assisting each parser in the current syntax rule to have an access to a common environment for their own semantic rules' future needs.

Static synthesized attributes are provided within the definitions of the AG combinators *terminal* and *empty* that define the terminals in the AG rules, and these attributes are passed upward with the end positions in a tree of type *leaf* only if the terminal successfully consumes an input token. Declarative synthesized and inherited rules are constructed with the help of the function *rule* by applying a desired user operation to a list of synthesized and/or inherited expressions from surrounding parsers. This function helps to find matching expressions from the *parser* and *nt* provided environment of attributes by threading some comparison factors through the current syntax-directed execution path as unevaluated instructions. These comparison factors are collected from the user-defined function's argument list of expressions either as *LHS* (i.e., when the current parser's attribute is used in the semantics) or as any other parser's unique id in the syntax.

Using the combinators described above, general attribute grammars that we mentioned in this paper can be constructed as executable specifications of language descriptions.

## V. MODELING UNIFICATION-BASED FORMALISMS

As with CFGs, unification-based formalisms [18] are language describing machines with added capabilities to impose linguistically motivated restrictions. These formalisms use an information-domain of *feature structures*, which are partial functions from features to their values where the values can be atomic or feature structures. A NL phrase is associated with a feature structure to guarantee its category and properties. *Unification* is a declarative process of restrictive combinations of information from two feature structures to form a new structure, and if the combination fails then the formalism does not recognize the current input. Using unification, phrases such as *a group of boys are attending the seminar for boy scouts* is accepted, while *James are sleeping* is rejected.

The general notation of AGs that we introduced earlier can be utilized for modular and declarative modeling of unification. We can compare *feature structures* with a set of synthesized or inherited expressions. The unification takes place by comparing evaluated values of these attributes according to the restriction that is needed to model linguistic correctness. We introduce a special-purpose synthesized attribute *Kill* of type *bool* for all syntax symbols in an AG. The one-pass syntax and semantic analysis algorithm is modified so that when the set of synthesized expressions for a non-terminal has an expression dedicated for the *Kill* attribute, and when this attribute is evaluated to *True* then the entire current result is discarded. This step is required to ensure that if a condition or restriction in the semantics fails then the current parse should not be a member of the result. When the expression for *Kill Bool* is mapped onto the current result, a predefined operation *nomatch* tries to determine whether all input argument expressions are evaluated to the same value or not:

```
nomatch(e1 : e2 : inputExps, environment)
= do let att1 ← e1(environment), att2 ← e2(environment)
      if att1/ = att2 then return Kill True
      else return nomatch(inputExps, environment)
```

```
nomatch([e1, e2], environment)
= do let att1 ← e1(environment), att2 ← e2(environment)
      if att1/ = att2 then return Kill True else return Kill False
```

These input expressions form restricted dependencies that can be used to impose linguistically motivated conditions in conjunction with compositional semantics. The following simple directly executable AG shows that phrases such as *moons that spin* is recognized as a grammatically correct sentence, but if we test the input is the phrase *moons that spins* then the parse fails because of the grammatical disagreement that sets *Kill True* for the sentence.

```
sent = memoize Sent
(parser (nt termph S1 * > nt relpro S2 * > nt vbph S3)
 [rule_s Kill OF LHS EQ nomatch
 [synthesized VAL OF S1, synthesized VAL OF S3]
 ,rule_s SENT_VAL OF LHS EQ apply_termphrase
 [synthesized TERMPH_VAL OF S1
 ,synthesized RELPRO_VAL OF S2
```

```

, synthesized VERBPH_VAL OF S3]])

termph = memoize Termph (terminal (term "moons")
 [TERMPH_VAL set_of_moons, VAL "plural"]
 <|> terminal (term "planets")
 [TERMPH_VAL set_of_planets, VAL "plural"])

vbph = memoize Vbph (terminal (term "spin")
 [VERBPH_VAL set_of_spin, VAL "plural"]
 <|> terminal (term "spins")
 [VERBPH_VAL set_of_spin, VAL "singular"])

relpro = memoize Relpro
 (terminal (term "that") [RELPRO_VAL intersect]
 <|> terminal (term "who") [RELPRO_VAL intersect])

```

Note that the other semantic rules are constructed based on the set-theoretic version of Montague’s compositional semantics. Our declarative notation for compositional semantics allows us to establish unbounded dependency that exists in relative clause sentences. According to Montague, relative pronouns (*relpro*) act as a conjunction (i.e. *and*) for two surrounding phrases *p* and *q* (i.e.,  $and = \lambda p \lambda q (\lambda z (p z \wedge q z))$ ). In the set-theoretic version, the *and* is defined as set intersection between two phrases, which are computed as sets too. In the above example, relative pronouns (*that*, *who*) have a synthesized attribute *intersect*, which is a function that expects two sets as input to perform the set intersection operation. This attribute is propagated upwards to the *sent*’s definition as *RELPRO\_VAL*. The *sent* computes the final *SENT\_VAL* through an *apply\_termphrase* function that provides two sets (*TERMPH\_VAL* and *VERBPH\_VAL*) to *RELPRO\_VAL* from two surrounding phrases - term phrase and verb phrase.

In [19] Correa systematically compared properties of attribute grammars and unification grammars, and demonstrated that attribute grammars are more general than unification grammars in terms of expressive power and computational efficiency. He also mentioned difficulties in implementing generalize attribute grammars because of problems related to attribute evaluations. As our parsing strategy is non-strict, our attribute evaluation order is demand driven and untangled by nature. Moreover, unlike unification-based notations, we can refer to unevaluated expressions as attributes in semantic dependencies, which enriches the overall declarativeness and modularity of the language description.

## VI. ACCOMMODATING PHENOMENA BEYOND CONTEXT-FREE

It has been well-documented that a few instances of natural language (e.g., some sentences from Dutch or Swiss German) may not be accommodated by context-free grammars; rather they need a stronger formalisms that are often referred to as mildly context-sensitive grammars (MCSGs) [20]. MCSGs are strictly stronger than CFGs and strictly weaker than context sensitive grammars in terms of generative power. Many formal formalisms strictly or weakly satisfy characteristics of MCSG such as tree adjoining grammar (TAG), combinatory categorial grammar (CCG), linear indexed grammar (LIG) etc. (see [21] for the proof of their equivalence). Common characteristics that define MCSG formalisms include 1) that they contain

all context-free languages, 2) that the membership problem is solvable in polynomial time, 3) that a member language grows constantly, and 4) that these formalisms contain the following non context-free languages [22] -

- L1. *multiple agreement*  
 $\{a^n b^n c^n \mid n \geq 0\}$
- L2. *crossed agreement*  
 $\{a^n b^m c^n d^m \mid n, m \geq 0\}$
- L3. *duplication*  
 $\{ww \mid w \in \{a, b\}^*\}$

We now show how to take advantage of arbitrary attribute dependencies to model above languages. As AGs generate languages that are founded on context free grammars, it is not directly possible to generate the above non context-free languages, but it is possible to non-strictly filter out sentences that do not belong to the target language by imposing semantic restrictions that guarantee only acceptance of members of L1, L2, and L3. A CFG  $S ::= As^* > Bs^* > Cs$ ,  $As ::= As^* > a|a$ ,  $Bs ::= Bs^* > b|b$ ,  $Cs ::= Cs^* > c|c$  that generates a set of sentences (in the form of any number of *as* followed by any number of *bs* then any number of *cs*) can also potentially include sentences generated by L1. But, if we can enforce that the *As*, *Bs*, and *Cs* can be expanded or recursively re-write themselves exactly the same number of times (i.e., *n*) then this CFG is restricted to a grammar only for L1. This restriction is shown with the following example AG by checking *As*, *Bs*, and *Cs*’s *count* attribute value. Each time the *As*, *Bs*, or *Cs* accepts a token (e.g., *a*, *b*, or *c* respectively), their synthesized *count* attribute’s values increase, and at the root non-terminal *S* these *counts* are compared with *nomatch*. If any mismatch is found then the *S*’s *Kill* attribute is set to *True*, and the current parse is entirely discarded.

```

s = memoize Ss
 (parser (nt as S0 * > nt bs S1 * > nt cs S2 )
 [rule_s Kill OF LHS EQ nomatch
 [synthesized Count OF S0, synthesized Count OF S1
 , synthesized Count OF S2]])

as = memoize As (parser (nt as S0 * > nt term_a S1 )
 [rule_s Count OF LHS EQ increment
 [synthesized Count OF S0]]
 <|> parser (nt term_a S1)
 [rule_s Count OF LHS EQ copy
 [synthesized Count OF S1]])

bs = memoize Bs (parser (nt bs S0 * > nt term_b S1 )
 [rule_s Count OF LHS EQ increment
 [synthesized Count OF S0]]
 <|> parser (nt term_b S1)
 [rule_s Count OF LHS EQ copy
 [synthesized Count OF S1]])

cs = memoize Cs (parser (nt cs S1 * > nt term_c S1 )
 [rule_s Count OF LHS EQ increment
 [synthesized Count OF S1]]
 <|> parser (nt term_c S1)
 [rule_s Count OF LHS EQ copy
 [synthesized Count OF S1]])

term_a = memoize TA (terminal (term "a") [Count 1])
term_b = memoize TB (terminal (term "b") [Count 1])
term_c = memoize TC (terminal (term "c") [Count 1])

```

Note that in the above example *increment* and *copy* are user-defined functions (like *nomatch*) that increment and make a copy of an attribute value whenever demanded. Using a similar concept, an AG for L2 can be constructed declaratively by using another user defined function *crossmatch* to impose

cross serial dependencies. The *crossmatch*'s four argument expressions' *count* values are now used to decide whether the root non-terminal's *Kill* value is *True* or *False*:

```
crossmatch([e1, e2, e3, e4], environment)
= do let count1 ← e1(environment), count2 ← e2(environment)
      count3 ← e3(environment), count4 ← e4(environment)
  if count1 == count3 and count2 == count4
  then return Kill False else return Kill True
```

The application of *crossmatch* in AG the for the *L2* can be shown with the following partial executable specification:

```
s = memoize Ss
(parser (nt as S0 *> nt bs S1 *> nt cs S2 *> nt ds S3 )
 [rule_s Kill OF LHS EQ crossmatch
 [synthesized Count OF S0, synthesized Count OF S1
 ,synthesized Count OF S2, synthesized Count OF S3]]).....
```

To model the duplicate or 2-place copy language *L3* with our notation of AGs we can use a stack or list-like data structure as an attribute type. The backbone CFG that potentially can contain members of *L3* along with other unwanted sentences can be represented as follows:

$$S ::= W * > W$$

$$W ::= a * > W \mid b * > W \mid \epsilon$$

The definition for *W* satisfies generating  $\{a, b\}^*$ , but one *W* following another does not guarantee generating *L3*. If *W* had an attribute as a list of indicators representing all accepted *as* and *bs* by the current *W*, and if we can set the root *S*'s *Kill* value to *True* whenever two lists of indicators for two consecutive *W*'s are not identical then unwanted sentences will be discarded. We can accomplish this objective with our notation for AG as shown with the partially-complete example below. Assume that new user function *nomatchList* checks equality of two operand lists, and *addToList* attaches the first argument to the second, which is a list originated as an empty list from the alternative *empty*.

```
s = memoize Ss (parser (nt w S0 *> nt w S1)
 [rule_s Kill OF LHS EQ nomatchList
 [synthesized List OF S0, synthesized List OF S1]])
w = memoize W (parser (nt term_a S0 *> nt W S1 )
 [rule_s Count OF LHS EQ addToList
 [synthesized Val OF S0, synthesized List OF S1]]
<|> parser (nt term_b S0 *> nt W S1 )
 [rule_s Count OF LHS EQ addToList
 [synthesized Val OF S0, synthesized List OF S1]])
<|> terminal (empty) [List []]).....
```

The above example specifications of executable AGs for languages *L1*, *L2*, and *L3* could also have been constructed differently by utilizing different variations of semantics constrains by introducing new inherited and/or synthesized attributes.

## VII. CONCLUDING COMMENTS

We have demonstrated how a declarative notation for general AGs can be used to create executable specifications to modularly prototype many NLP tasks. Our approach is founded on a top-down parsing technique that accommodates any form of CFG (including left recursive grammars) coupled with declarative semantics using the syntactic constituents' attributes. The overall evaluation technique is non-strict and

treats attributes as unevaluated functions, and as a result arbitrary dependencies between syntactic components can be described in the semantics. We have shown that restrictions can be imposed using special-purpose attributes and customizable semantic functions to model natural language properties such as unification, and phenomena that cannot be accommodated with only CFGs. In the future we plan to investigate further incorporating dependencies involving inherited attributes along with synthesized attributes to construct general-purpose natural language interfaces where compositional semantics would compute meanings. We are intending to investigate many other forms of natural language disambiguation using our declarative restrictive capability. We also believe that our transparent syntax-semantic interface can be used effectively for syntax-based machine translation applications.

## REFERENCES

- [1] F. C. N. Pereira and D. . Warren, "Define clause grammars for language analysis." *Artificial Intelligence*, vol. 13, 1980.
- [2] P. Hudak, J. Peterson, and J. Fasel, "A gentle introduction to Haskell 98," Tech. Rep., 1999.
- [3] P. Wadler, "Monads for functional programming," *1st Spring School on Advanced FP*, vol. 925, pp. 24 – 52, 1995.
- [4] R. Frost, R. Hafiz, and P. Callaghan, "Modular and efficient top-down parsing for ambiguous left-recursive grammars." *ACL-IWPT*, pp. 109 – 120, 2007.
- [5] R. Frost and R. Fortier, "An efficient denotational semantics for natural language database queries." in *Applications of NLDB*, 2007, pp. 12–24.
- [6] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling, Vol. 1, Parsing*. Prentice Hall, 1972.
- [7] D. Knuth, "Semantics of context-free languages," *Theory of Computing Systems, Springer New York*, vol. 2, no. 2, pp. 127–145, 1968.
- [8] O. De Moor, K. Backhouse, and D. Swierstra, "First-class attribute grammars," in *3rd Workshop on AGs and their Applications*, 2000.
- [9] M. Tienari, "On the definition of attribute grammar," *Semantics-Directed Compiler Generation*, vol. 94, pp. 408 – 414, 1980.
- [10] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper, "Higher order attribute grammars," in *PLDI*. ACM, 1989, pp. 131–145.
- [11] P. Wadler, "How to replace failure by a list of successes." in *FP languages and computer architecture*, vol. 201, 1985, pp. 113 – 128.
- [12] R. Frost and J. Launchbury, "Constructing natural language interpreters in a lazy functional language," *The Computer Journal*, vol. 32(2), 1989.
- [13] G. Hutton and E. Meijer, "Monadic parser combinators." *J. Funct. Program.*, vol. 8(4), pp. 437 – 444, 1998.
- [14] R. Frost, R. Hafiz, and P. Callaghan, "Parser combinators for ambiguous left-recursive grammars." *ACM-PADL*, pp. 167–181, 2008.
- [15] P. Norvig, "Techniques for automatic memoization with applications to context-free parsing." *Computational Linguistics*, vol. 17(1), pp. 91 – 98, 1991.
- [16] M. Tomita, *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1986.
- [17] R. Hafiz and R. A. Frost, "Lazy combinators for executable specifications of general attribute grammars." in *ACM-PADL*, 2010, pp. 167–182.
- [18] S. Shieber, "An introduction to unification-based theories of grammar," *CSLI Lecture Notes Series, University of Chicago Press.*, 1986.
- [19] N. Correa, "Attribute and unification grammar: A review and analysis of formalisms," *Annals of Mathematics and AI*, vol. 873-105, 1993.
- [20] A. K. Joshi, "Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions?" *NL Parsing, Cambridge University Press*, 1985.
- [21] k. Vijay-Shanker and D. J. Weir, "The equivalence of four extensions of context-free grammars," *Theory of Computing Systems*, vol. Volume 27 Issue 6, pp. 511–546, 1994.
- [22] M. Kudlek, C. Martn-Vide, A. Mateescu, and V. MitranSource, "Contexts and the concept of mild context-sensitivity," *Linguistics and Philosophy*, vol. Vol. 26, No. 6 ., pp. 703–725, 2003.