

Executable Specifications of Fully General Attribute Grammars with Ambiguity and Left-recursion

Rahmatullah Hafiz

School of Computer Science, University of Windsor, Canada.

Abstract. A top-down parsing algorithm has been constructed to accommodate any form of ambiguous context-free grammar, augmented with semantic rules with arbitrary attribute dependencies. A memoization technique is used with this non-strict method for efficiently processing ambiguous input. This one-pass approach allows Natural Language (NL) processors to be constructed as executable, modular and declarative specifications of Attribute Grammars.

1 Introduction

In an Attribute Grammar (AG) [6], syntax rules of a context-free grammar (CFG) are augmented with semantic rules to describe the meaning of a context-free language. AG systems have been constructed primarily as compilable parser-generators for formal languages. Little evidence could be found where fully-general AGs have been used for declaratively specifying directly-executable specifications of NLS. None of the existing approaches support arbitrary attribute dependencies (including dependencies from right) in semantics and general context-free syntax (including ambiguity and left-recursion) altogether in one-pass within a single top-down system. Basic top-down parsers do not terminate while processing left-recursion, and are normally inefficient while parsing exponentially ambiguous grammars (e.g., $S ::= SSa|\epsilon$, on input “aaa...” [1]).

A top-down approach that supports executable and declarative specifications of AGs with unconstrained syntax and semantics, offers following advantages:

- a. Language descriptions can be specified and executed directly without worrying about syntax-semantics evaluation method and order.
- b. Individual parts of AGs can be efficiently tested separately. Modularity assists with systematic and incremental development.
- c. Accommodating ambiguity is vital for some domain e.g. NLP.
- d. Transforming a left-recursive CFG to a weakly equivalent non-left-recursive form may introduce loss of parses and lack of completeness in semantic [8, 4].
- e. Arbitrary attribute dependencies provide unrestricted and declarative construction of complex semantic (e.g., set-theoretic Montague semantics with CFG).
- f. If implemented using a modern functional language, then benefits such as purely-functional, declarative and non-strict environment are readily available.

2 Our Contributions

This paper describes an extension of our previous work by accommodating the executable specifications of arbitrary semantic rules coupled with general syntactic rules. In [4, 5], we have shown how top-down parsers can be constructed as executable specifications of general CFGs defining the language to be parsed:

- 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 condition “*left – recursive count* of parser at $j > \#input - j + 1$ ” succeeds.
- 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 .

Our extended parser-application is a mapping from an input’s start position to a set of end positions with tree structures. We also thread attribute values along with the start and end positions so that they are available for any dependencies that are specified in semantic rules. These attribute values are defined in terms of *expressions* (as our method is referentially transparent and non-strict) that represent arbitrary operations on syntax symbols’ attributes, which are computed from the surrounding *environment* when required.

The structured and lazily computed *result* of parser-applications allow us to establish full call-by-need based dependencies between attributes - including inherited dependencies from the right and top. For example, when a parser p_i with syntax $p_i ::= p_m p_n$, is applied to input position 1 and successfully ends at position 5, we represent one of p_i ’s input/output relations (as a result) as:

$$SubNode\ p_i\ inh_{p_i1} = syn_{p_i5} [..Branch\ [SubNode\ p_m\ (inh_{p_m1}, syn_{p_m3}) \\ , SubNode\ p_n\ (inh_{p_n3}, syn_{p_n5})]..]$$

where, 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. Now arbitrary semantics can be formed with non-strict expressions e.g., $inh_{p_m i} \leftarrow f(inh_{p_n j}, syn_{p_i k})$, where f is a desired operation on attributes.

We have achieved our objective by defining a set of combinators for modular, declarative and executable language processors, which are similar to textbook AG notation. Our combinators (e.g., `<|>` and `*>` correspond to `orelse` and `then`, `rule_s` and `rule_i` for synthesized and inherited semantic rules, `parser` and `nt` for complete AG rules and non-terminals etc.) are pure, higher-order and lazy functions that ensure fully-declarative executable specifications. We have implemented our method in a lazy functional language Haskell. We execute syntax and semantics in one-pass in polynomial time using a memoization technique to ensure that results for a particular parser at a particular position are computed at most once, and are reused when required. We represent potentially exponential results for ambiguous input in a compact and shared tree structure to achieve polynomial space. The *memo-table* is systematically threaded through parser executions using state-monad [7].

3 An Example

We illustrate our approach with the `repmax` example [2,3], which we have extended to accommodate ambiguity, left-recursion and arbitrary attribute dependencies. Our goal is to parse inputs such as “1 5 2 3 2” with an ambiguous left-recursive CFG $tree ::= tree\ tree\ num \mid num$, $num ::= 1|2|3|4|5|...$, and to extract all possible trees (for syntactic correctness) with all terminals replaced by maximum value of the sequence. The AG that specifies this problem is as follows (where \uparrow and \downarrow represent synthesized and inherited attributes respectively):

$$\begin{aligned}
 start(S_0) & ::= tree(T_0) \{RepVal.T_0 \downarrow = MaxVal.T_0 \uparrow\} \\
 tree(T_0) & ::= tree(T_1)\ tree(T_2)\ num(N_1) \\
 \{ & MaxVal.T_0 \uparrow = Max(MaxVal.T_1 \uparrow, MaxVal.T_2 \uparrow, MaxVal.N_1 \uparrow), \\
 & RepVal.T_1 \downarrow = RepVal.T_0 \downarrow, RepVal.T_2 \downarrow = RepVal.T_0 \downarrow, \\
 & RepVal.N_1 \downarrow = RepVal.T_0 \downarrow\} \\
 & \mid num(N_2) \{MaxVal.T_0 \uparrow = MaxVal.N_2 \uparrow, RepVal.N_2 \downarrow = RepVal.T_0 \downarrow\} \\
 num(N_0) & ::= 1 \{MaxVal.N_0 \uparrow = 1\} \mid 2 \{MaxVal.N_0 \uparrow = 2\} \dots
 \end{aligned}$$

According to this AG, there are two outputs for the input sequence “1 5 2 3 2”:



Using our combinators and notation, an executable specification of the general AG above can be constructed declaratively in Haskell as follows:

```

start = memoize Start parser (nt tree T0) [rule_i RepVal OF T0 ISEQUALTO findRep
                                         [synthesized MaxVal OF T0]]
tree  = memoize Tree  parser (nt tree T1 *> nt tree T2 *> nt num T3)
      [rule_s MaxVal OF LHS ISEQUALTO
       findMax [synthesized MaxVal OF T1, synthesized MaxVal OF T2, synthesized MaxVal OF T3]
       ,rule_i RepVal OF T1 ISEQUALTO findRep [inherited RepVal OF LHS]...]
<|>   parser (nt num N1)
      [rule_i RepVal OF N1 ISEQUALTO findRep [inherited RepVal OF LHS]
       ,rule_s MaxVal OF LHS ISEQUALTO findMax [synthesized MaxVal OF N1]]
num   = memoize Num  terminal (term "1") [MaxVal 1] <|> ... <|> terminal (term "5") [MaxVal 5]

```

When `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 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 parser-name, unique-id pairs e.g. `(Tree, T1)`) with inherited and synthesized attributes:

```

Tree START at : 1 ; Inherited atts: T0 RepVal 5 T0 RepVal 1
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))...

```

Our semantic rules declaratively define arbitrary actions on the syntax symbols' attributes. For example, the second semantic 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.

4 Concluding Comments

The goal of our work is to provide a framework where general CFGs (including ambiguous left-recursion) can be integrated with semantic rules with arbitrary attribute dependencies as directly-executable and modular specifications. Our underlying top-down parsing method is constructed with non-strict combinators for declarative specifications, and uses a memoization technique for polynomial time and space complexities. In the future we aim to utilize grammatical and number agreements, and conditional restrictions for disambiguation. By taking the advantages of general attribute dependencies, we plan to efficiently model NL features that can be characterized by other grammar formalisms like unification grammars, combinatory categorical grammars and type-theoretic grammars. We have implemented our AG system in Haskell, and have constructed a Natural Language Interface based on a set-theoretic version of Montague semantic (more can be found at: cs.uwindsor.ca/~hafiz/proHome.html). We are constructing formal correctness proofs, developing techniques for detecting circularity and better error-recognition, and optimizing the implementation for practical uses. We believe that our work will help domain specific language developers (e.g. computational linguists) to build and test their theories and specifications without worrying about the underlying computational methods.

References

1. A. V. Aho and J. D. Ullman. *The Theory of Parsing, Vol I*. Prentice-Hall., 1972.
2. R. S. Bird. *Intro. to Functional Programming in Haskell*. Prentice Hall, 1998.
3. O. De Moor, K. Backhouse, and D. Swierstra. First-class attribute grammars. In *WAGA*, 2000.
4. R.A. Frost, R. Hafiz, and P. Callaghan. Modular and efficient top-down parsing for ambiguous left-recursive grammars. *10th IWPT, ACL.*, pages 109 – 120, 2007.
5. R.A. Frost, R. Hafiz, and P. Callaghan. Parser combinators for ambiguous left-recursive grammars. *10th PADL, ACM.*, 4902:167–181, 2008.
6. D. E. Knuth. Semantics of context-free languages. *Theory of Comp. Systems*, 2(2):127–145, 1968.
7. P. Wadler. Monads for functional programming. *First International Spring School on Advanced Functional Programming Techniques*, 925:24 – 52, 1995.
8. D.S. Warren. Programming the ptq grammar in xsb. In *Workshop on Programming with Logic Databases*, pages 217–234, 1993.