Lazy Combinators for Executable Specifications of General Attribute Grammars

Rahmatullah Hafiz and Richard A. Frost
hafiz@uwindsor.ca and richard@uwindsor.ca
Department of Computer Science
University of Windsor, Ontario, Canada

at

PADL'2010
Madrid, Spain

The Problem

- Knowledge of parsing techniques and semantic evaluation methods are required in order for software developers to create natural language (NL) processors, thereby restricting the creation of NL interfaces to many applications.

- One solution is to allow language processors to be created as executable specifications of grammars annotated with semantic rules. The most modular approach is to use top-down parsing.

The Problem (cont.)

However, naïve algorithms for top-down parsing:

1. Do not non-terminate for context-free syntax that includes direct/ indirect left-recursion
   - addressed by Frost, Hafiz and Callaghan (PADL 2008)

2. Require exponential time and space for ambiguous grammars e.g., grammars for NL

3. Do not provide support for general attribute relationships (including the use, by a parser for construct *p*, of inherited attributes associated with syntactic constructs “to the right” of *p* on the right hand side of syntactic rules.

4. 2 and 3 are the primary focus of this paper

Importance of the Problem

- Accommodating ambiguity is essential as natural languages have ambiguous grammars.
- Transforming a left-recursive grammar to a weakly equivalent non-left-recursive form can introduce loss of parses and difficulty in specifying semantics.
- Declarative semantics with arbitrary attribute dependencies provide unrestricted accommodation of NL semantics. For example, Montague style compositional semantics.

Importance of the Problem (cont.)

- Language developers can build executable specifications of language processors without worrying about syntax-semantics evaluation methods and order.

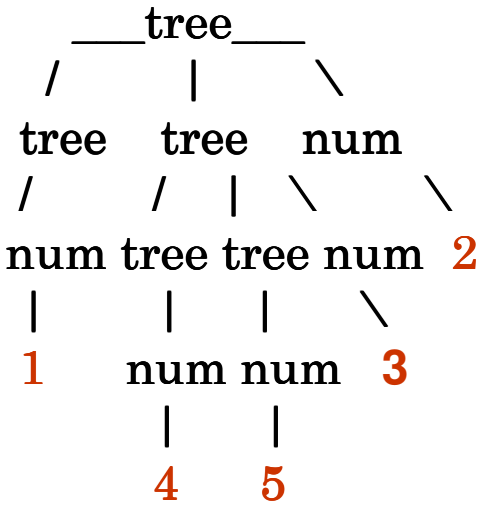
- Modularity allows individual parts of the specifications to be constructed and tested separately.

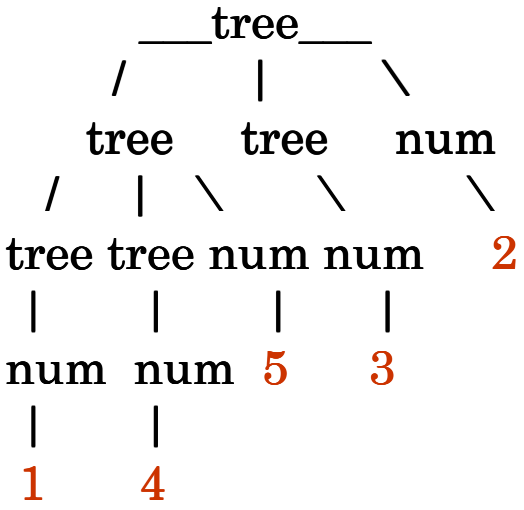
- Polynomial time and space are required for parsing highly ambiguous languages, such as NL, in real time.

An example of a general CFG

$start\ (S_0)$ ::= $tree(T_0)$
$tree(T_0)$ ::= $tree(T_1)\ tree(T_2)\ num(N_1)$
| $num(N_2)$
$num(N_0)$ ::= 1
| 2 .....

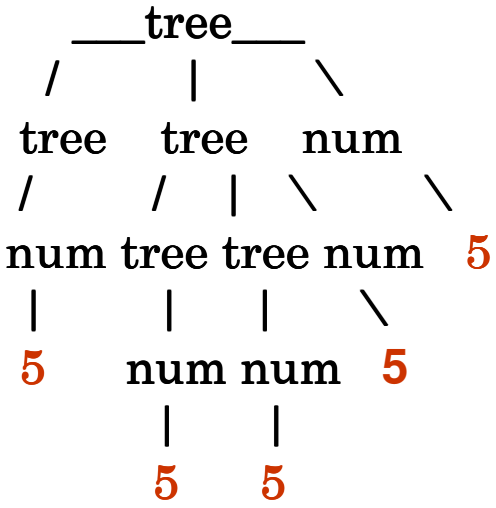
This left-recursive and ambiguous context-free syntax generates two ambiguous trees when applied to the input 1 4 5 3 2.

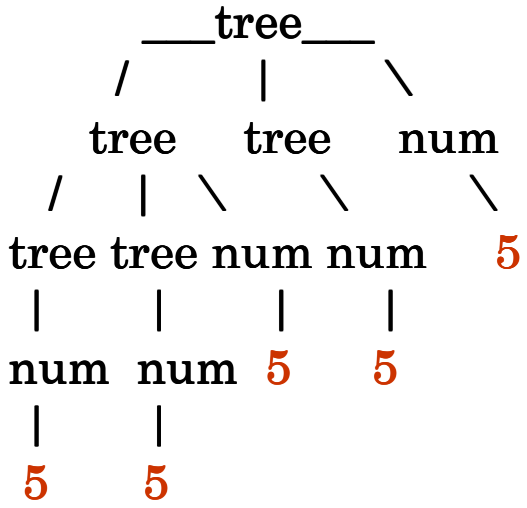




An example problem

Suppose that we want to replace all leaves of the trees with the max value of the input, giving:





One approach is to specify this problem as a language processing problem, and to define the solution using an executable attribute grammar.

An executable specification

```
start (S0)   ::= tree(T0)
  { RepVal.T0↓ = MaxVal.T0↑ }
tree(T0)     ::= tree(T1) tree(T2) num(N1)
  { MaxVal.T0↑ = max (MaxVal.T1↑,
                      MaxVal.T2↑,
                      MaxVal.N1↑)
  , RepVal.T1↓  = RepVal.T0↓
  , RepVal.T2↓  = RepVal.T0↓
  , RepVal.N1↓  = RepVal.T0↓
  }
             |    num(N2)
num(N0)   ::= 1{MaxVal.N0↑= 1}|2{MaxVal.N0↑=2}..
```

↓ = attributes propagating downwards i.e. **inherited attributes**
↑ = attributes propagating upwards i.e. **synthesized attributes**

Our objective is to have a program that is isomorphic with this grammar, i.e. an executable specification.

Our Solution

- We have constructed the first top-down parsing algorithm that supports executable specifications of fully general CFGs annotated with fully-general declarative semantic rules in polynomial time and space w.r.t the length of input.

- We have implemented the algorithm as a set of non-strict, purely functional combinators, i.e. higher-order functions:

  `*>` and `<|>` for sequencing and alternating rules

  `rule_i` and `rule_s` for synthesized and inherited rules

  `parser`, `nt` for complete AG rules

  `memoize` for converting parsers to memoized versions.

Solution (cont.)

Our parser combinators are functions that map the current start position to a set of end positions, where each end is mapped to a set of syntax trees. For example:

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

type Start/End = (Int,[(Instance, [Atts])])

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

Solution (cont.)

Memoization allows use of previously-computed results:

- The memo-table is threaded through parser executions using a state monad

```
type MemoTable = [(Label,[(Start,
                           (Context,Result))])]
type Result    = [((Start, End),[PTree Label])]
```

- lookup is performed, if no previous application, a new result is constructed and the memo-table is updated.
- groups local syntactic ambiguities and common semantic values under the current position in a newly-formed result.
- parsers pass up a reference/pointer of their memo-table entry to upper-level parsers, instead of the complete result.

Solution: General Syntax

The basic syntax analysis technique is based on Frost, Hafiz and Callaghan (2008):

- To accommodate direct left-recursion, a “left-recursive count” is used for the number of times a parser has been applied to an input position j. 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 the current parse path, which is used to determine the context in which the result was constructed at j – if the new context strictly subsumes the context for a saved result, then a new result is computed, otherwise the saved result is used.

Solution: General Semantics

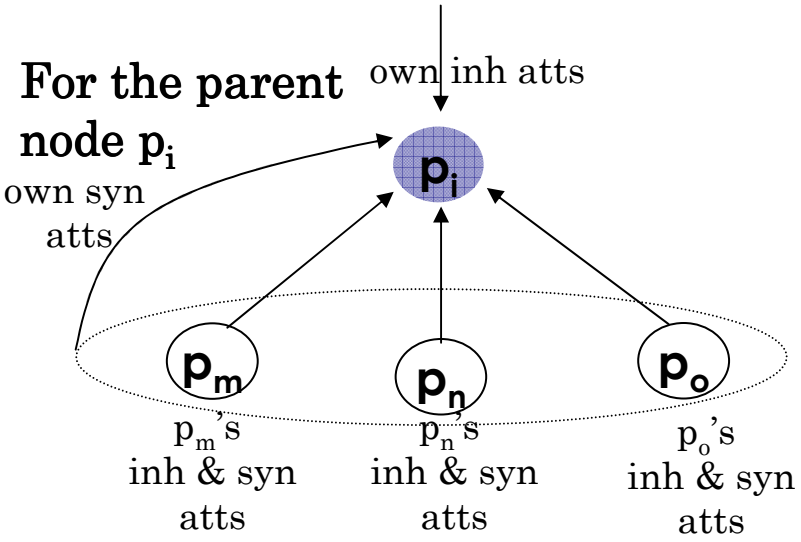
We have now extended the set of combinators to accommodate fully general semantics:

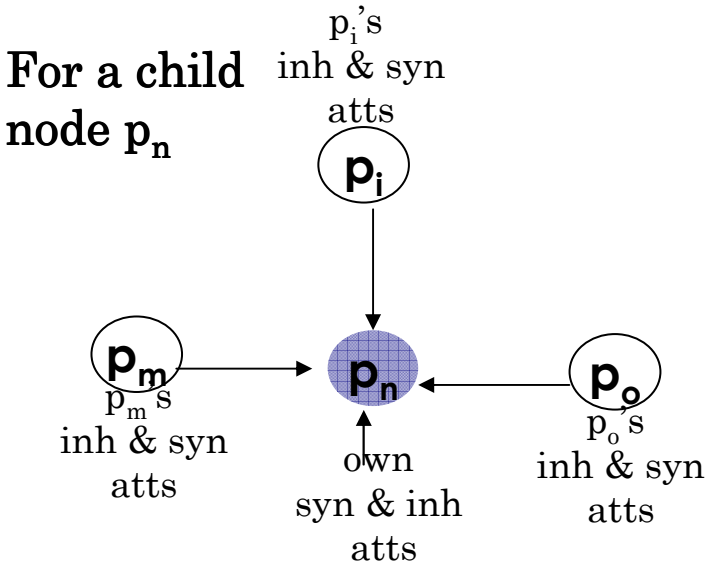
- Pure, non-strict, combinators simplified the accommodation of arbitrary dependencies including “inheritance from the right”
- Expressions are formed from an environment of potentially unevaluated attributes returned by the current parser, its predecessor, successors or siblings

Solution: General Semantics (cont.)

- Synthesized attributes are associated with parent nodes, and inherited attributes are associated with their child nodes

- To maintain the flow of attributes, in addition to being executed on the current Start and Context, a parser 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.





Haskell Combinators

Our combinators are implemented in Haskell.

A parser's alternative rules are formed with the combinator <|>.

- Accommodates alternative syntax with a list of semantic rules.
- Alternatives p and q are applied to the current position and the current context. The id and inherited attributes of the calling parser are passed down to the parsers for p and q. All results from p are passed to q (thereby avoiding duplication of work) and then their results merged together .

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

Haskell Combinators (cont.)

Parsers are sequenced with the combinator `*>`

- In p`*>`q, p is first applied to the current start position and the current context. Then `*>` causes p to compute its inherited attribute from the surrounding environment.

- Next, q is applied to the set of ends returned by p. q also computes inherited attributes from the calling and p's environment.

```
(*>) :: SeqType -> SeqType -> SeqType, where :

type SemRule = (Instance,(InsAttVals, Id) -> InsAttVals)

type SeqType = Id -> InsAttVals -> [SemRule] -> Result
                                              -> M ParseResult
```

Arbitrary Dependencies in Semantics (cont.)

Attribute computation rules are formed with a higher-order wrapper function `parser`,

- maps the current parser's synthesized rules to all ending points of the syntactic result.
- causes each parser in the syntax rule to pass down their inherited attributes for future use.

```
parser :: SeqType -> [SemRule] -> Id ->
          InsAttVals -> M ParseResult
```

The higher-order function `nt` causes parsers to pass down their own identification and a list of inherited attributes

- by applying the grouped expressions on a `parser`-provided environment that consists of the predecessor id's and surrounding parsers' synthesized and inherited attributes.

```
nt :: NTType -> Id -> SeqType
```

Our Example Executable Specification in Haskell

The Executable representation of the example attribute grammar specification of slide #8

```
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]
```

The Result of Executing the Specification

```
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))...................
```

Time Complexity w.r.t #input

For non-left recursive grammars, parsing complexity w.r.t. length of the input, n, is $O(n^3)$, and for left recursive grammars, the complexity is $O(n^4)$

When semantic evaluation is taken into account, the time complexity may increase depending on the complexities of the semantic actions.

Space Complexity w.r.t #input n

- Our compact representation of syntax trees allows the parser 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. 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.

An example application

A simple domain-specific NL interface:

- The Attribute Grammar is a fully-general CFG with 15 non-terminals and 32 AG rules.
- All syntax rules are associated with semantic rules which implement a subset of the set-theoretic version of Montague semantics extracted from Frost and Fortier (2007).
- We define a dictionary/ knowledge-base for syntactic categories and their meanings: 15 syntactic categories and 130 words.

An example application

```
dictionary =

  ("moons",         Cnoun,     [NOUNCLA_VAL set_of_moon]),

  ("atmospheric",   Adj,       [ADJ_VAL    set_of_atmospheric]),

  ("exist",         Intransvb, [VERBPH_VAL  set_of_things]),

  ("deimos",        Pnoun,     [TERMPH_VAL (test_wrt 20)]),

  ("person",        Cnoun, meaning_of nouncla "man or woman"),

  ("discoverer”,    Cnoun, meaning_of nouncla
                           "person who discovered something"),
```

An example application(cont.)

Example executable specifications of production rules:

```
jointermph = memoize Jointermph

 parser (nt jointermph S1 *> nt termphjoin S2 *> nt jointermph S3)
             [rule_s TERMPH_VAL OF LHS ISEQUALTO
                   apply_join [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
                      ISEQUALTO copy [synthesized TERMPH_VAL OF S4]]
```

An example application (cont.)

Answers hundreds of thousands of questions, e.g.

which moons that were discovered by hall orbit mars

⇒ **"phobos and deimos"**

every planet is orbited by a moon

⇒ **"false"**

how many moons were discovered by hall or kuiper

⇒ **"four"**

did hall discover deimos or phobos and miranda

⇒ **"no and yes"** ***** note: ambiguous results***

Related Work

- Kuno's (1965) "depth-imposing" algorithm, Nederhof and Koster's "cancellation- parsing" for DCG's (1993), Lickman's use of "fixed-points" (1995), and Johnson's integration of memoization with continuation-passing-style (1995), all terminate for left-recursion but have exponential complexity.

- Norvig (1991) and Frost's (1994) techniques for automatic memoization in parsing.

- Hutton and Meijer (1995) Monadic parser combinators.

- Swierstra et al. (1991 and 1998) and De Moor et al. (2000) "first-class attributes". JustAdd (Ekman, 2006) is an compiler-compiler AG system for Java.

- Frost (2002) – an Attribute Grammar programming environment. Also, YAG (Mcroy et al., 2003) in which AGs are used to correct partially-specified input.

Comparison with Related Work

- Our approach is directly executable and specifications do not need to be compiled
- Our top-down syntax-driven parsing strategy strictly preserves syntactic structures of general ambiguous CFGs
- Our approach accommodates fully declarative semantic with arbitrary dependencies for general syntax following original def. of AG
- Our memoization is specialized to perform extra tasks e.g. keeping track of non-terminals' context information, merging syntactic ambiguity, mapping and grouping attributes etc.
- Our approach differs from other NL processors by being a one-pass parsing system that can return either compactly-represented parse trees annotated with attribute values or just a set of final answers, according to what is demanded by the application.

Future Work

- Construct formal correctness proofs, and optimize the implementation w.r.t. grammar size.
- Use semantic rules to prune out non-sensible parses.
- Model NL features that can be characterized by other grammar formalisms.

Project Website

X-SAIGA – E**x**ecutable **S**pecif**I**cations of **G**rammars

`cs.uwindsor.ca/~hafiz/proHome.html`

A version of demo code can be found at:

`http://cs.uwindsor.ca/~hafiz/fullAg.html`