Attribute grammars

Course "Software Language Engineering"

University of Koblenz-Landau
Department of Computer Science
Ralf Lämmel
Software Languages Team
Overview

Parse tree

Attributed parse tree
Motivating example: Semantic analyses for a DSL
Semantic analyses for DSL for finite state machines

state S1;
state S2;
state S3;
trans S1 -> S2;
trans S2 -> S1;
trans S2 -> S3;

[C] 2009 Görel Hedin
This slide was taken from G. Hedin's GTTSE 2009 tutorial.

[http://dx.doi.org/10.1007/978-3-642-18023-1_4 and accompanying slides, “An Introductory Tutorial on JastAdd Attribute Grammars” by Görel Hedin]
DSL support for name resolution / analysis

Resolve source/target links and transitions.

[SLE course material and accompanying slides, "An Introductory Tutorial on JastAdd Attribute Grammars" by Görel Hedin]
DSL support for reachability analysis

state S1;
state S2;
state S3;
trans S1 -> S2;
trans S2 -> S1;
trans S2 -> S3;

Reachability

\{S1, S2, S3\}
\{\}
\{S1, S2, S3\}

[http://dx.doi.org/10.1007/978-3-642-18023-1_4 and accompanying slides, “An Introductory Tutorial on JastAdd Attribute Grammars” by Görel Hedin]
Overview

Attribute grammar

- context-free grammar
- attribute declarations per nonterminals
- semantic rules for relating attributes in derivation tree

Applications
- Semantic analysis (e.g., name analysis)
- Language translation (as in compilation)
A simple attribute grammar example due to D. Knuth.
### Binary to decimal conversion

<table>
<thead>
<tr>
<th>Binary</th>
<th>→</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>→</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>→</td>
<td>1</td>
</tr>
<tr>
<td>1 0</td>
<td>→</td>
<td>2</td>
</tr>
<tr>
<td>1 1</td>
<td>→</td>
<td>3</td>
</tr>
<tr>
<td>1 0 0</td>
<td>→</td>
<td>4</td>
</tr>
<tr>
<td>1 0 1</td>
<td>→</td>
<td>5</td>
</tr>
<tr>
<td>1 0 1 . 0 1</td>
<td>→</td>
<td>5 . 25</td>
</tr>
</tbody>
</table>
Context-free grammar of binary numbers (EBNF)

```
number  = bit+ ("." bit+)?
bit     = "0" | "1"
```
Context-free grammar of binary numbers (BNF)

```
number ::= bits rest
bits ::= bit
bits ::= bit bits
bit ::= "0"
bit ::= "1"
rest ::= ε
rest ::= "." bits
```
Binary-to-decimal conversion

Attribute numbers, bit sequences, bits with decimal values.

Use extra helper attributes for bit position and length.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>101.01</td>
<td>5.25</td>
</tr>
</tbody>
</table>
## Attributes for binary-to-decimal conversion

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Attribute</th>
<th>Category</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Val</td>
<td>syn</td>
<td>float</td>
</tr>
<tr>
<td>bits</td>
<td>Pos</td>
<td>inh</td>
<td>integer</td>
</tr>
<tr>
<td>bits</td>
<td>Len</td>
<td>syn</td>
<td>natural</td>
</tr>
<tr>
<td>bits</td>
<td>Val</td>
<td>syn</td>
<td>float</td>
</tr>
<tr>
<td>bit</td>
<td>Pos</td>
<td>inh</td>
<td>integer</td>
</tr>
<tr>
<td>bit</td>
<td>Val</td>
<td>syn</td>
<td>float</td>
</tr>
<tr>
<td>rest</td>
<td>Val</td>
<td>syn</td>
<td>float</td>
</tr>
</tbody>
</table>
Inherited vs. synthesized attributes

Intuitions

- Inherited attributes are passed down the tree.
- Synthesized attributes are passed up the tree.

Consider attributes per rule $X_0 ::= X_1 \ldots X_n$:

- Defined attributes: $\text{SYN}(X_0)$, $\text{INH}(X_1)$, $\ldots$, $\text{INH}(X_n)$
- Applied attributes: $\text{INH}(X_0)$, $\text{SYN}(X_1)$, $\ldots$, $\text{SYN}(X_n)$

There must be semantic rules for each production such that defined attributes are actually “defined” in terms of applied ones.
Semantic rules per production
Semantic rules per production

number ::= bits rest

bits.Pos = bits.Len - 1
number.Val = bits.Val + rest.Val
Semantic rules per production

\[
\text{bit ::= } "0" \\
\text{bit.Val} = 0
\]

\[
\text{bit ::= } "1" \\
\text{bit.Val} = 2 \ ^ \ \text{bit.Pos}
\]
Semantic rules per production

\[
\text{bits} ::= \text{bit} \\
\text{bit}.\text{Pos} = \text{bits}.\text{Pos} \\
\text{bits}.\text{Val} = \text{bit}.\text{Val} \\
\text{bits}.\text{Len} = 1
\]

\[
\text{bits}_0 ::= \text{bit} \text{bits}_1 \\
\text{bit}.\text{Pos} = \text{bits}_0.\text{Pos} \\
\text{bits}_1.\text{Pos} = \text{bits}_0.\text{Pos} - 1 \\
\text{bits}_0.\text{Val} = \text{bit}.\text{Val} + \text{bits}_1.\text{Val} \\
\text{bits}_0.\text{Len} = \text{bits}_1.\text{Len} + 1
\]
Semantic rules per production

\[
\text{rest ::= } \varepsilon \\
\text{rest.Val} = 0
\]

\[
\text{rest ::= } \cdot \text{ bits} \\
\text{rest.Val} = \text{bits.Val} \\
\text{bits.Pos} = -1
\]
Attribute evaluation
Assume that inherited attributes of start symbol are defined.

Instantiate semantic rules with node identities for nonterminal nodes.

Solve the equational system by substitution.

An attribute grammar is well-formed if the system has a unique solution. (In particular, cyclic dependencies must not be present.)
Equational system

```
... b.Pos = b.Len - 1
a.Val = b.Val + c.Val

h.Pos = b.Pos
h.Val = h.Val + d.Val
b.Val = h.Val + d.Val
b.Len = d.Len + 1

...```

Diagram:

```
... h:bit i:bit j:bit k:bit l:bit
  /   /   /   /
  1   0   1   .
  |   |   |   |
  f:bits d:bits c:rest b:bits
    |     |     |
    |     |     |
    |     |     |
    1     1   0   1
```

More on evaluation

- Use work-list algorithm to defer rules until applicable.

Special cases:

- S-attributed: bottom-up evaluation
- L-attributed: evaluation proceeds from left to right.
- One pass: defined attributes are computable from applied ones in some order.
- Multi pass: a statically known sequence of passes can be used.
An S-attributed variation

number ::= bits rest
bits ::= bit
bits₀ ::= bits₁ bit
bit ::= "0"
bit ::= "1"
rest ::= ε
rest ::= "." bits

number.Val = bits.Val + rest.Val
bits.Val = bit.Val
bit.Val = 0
bit.Val = 1
rest.Val = 0
rest.Val = bits.Val / 2 ^ bits.Len
bits.Len = 1
bits₀.Len = bits₁.Len + 1
Implementation of attribute grammars
Overall options

- Use designated attribute grammar system
  - Yacc (?), ANTLR (?), Eli, JastAdd, ...

- Use expressive, declarative language
  - Prolog with constraints, Haskell with laziness

- Implement in free-wheeling code
  - ANTLR/Java hybrid, Recursive descent
Binary-to-decimal number conversion with ANTLR

https://github.com/slecours/slecours/tree/master/sources/attributeGrammars/knuth/antlr
Mapping attribute grammars to Haskell with laziness
Mapping attribute grammars to Haskell with laziness

Nonterminals become functions.

- One equation per production.

Attributes become function arguments and results – tuples thereof.

- Inherited attributes make up the argument tuple.
- Synthesized attributes make up the result tuple.
Algebraic datatypes instead of BNF

-- Non-negative binary numbers with optional binary places
data Number = Number Bits Rest

-- Non-empty bit sequences
data Bits = OneBit Bit | MoreBits Bit Bits

-- Single bits
data Bit = Zero | One

-- Binary places, if any
data Rest = None | Places Bits

number ::= bits rest
bits ::= bit
bits ::= "\bits" bits
bit ::= "0"
bit ::= "1"
rest ::= ε
rest ::= "." bits
One function per nonterminal

```haskell
evalNumber :: Number -> Float
evalNumber (Number bits rest) = val0
    where
        (len1, val1) = evalBits bits pos1
        pos1 = len1 - 1
        val2 = evalRest rest
        val0 = val1 + val2
```

```
number ::= bits rest

bits.Pos = bits.Len - 1
number.Val = bits.Val + rest.Val
```
One function per nonterminal

```plaintext
evalBit :: Bit -> Int -> Float

- evalBit Zero _pos = 0
- evalBit One pos = 2 ^^ pos
```
One function per nonterminal

\[
eval\text{Bits} :: \text{Bits} \to \text{Int} \to (\text{Int}, \text{Float})
\]

\[
eval\text{Bits} (\text{OneBit} \ bit) \ pos = (1, \eval\text{Bit} \ bit \ pos)
\]

\[
eval\text{Bits} (\text{MoreBits} \ bit \ bits) \ pos0 = (\text{len0, val0})
\]

where

\[
\text{val1} = \eval\text{Bit} \ bit \ pos0
\]
\[
(\text{len1, val2}) = \eval\text{Bits} \ bits \ pos1
\]
\[
\text{pos1} = \pos0 - 1
\]
\[
\text{len0} = \text{len1} + 1
\]
\[
\text{val0} = \text{val1} + \text{val2}
\]

bits ::= bit

bit.Pos = bits.Pos
bit.Val = bits.Val
bit.Len = 1

bits_0 ::= bit bits_1

bit.Pos = bits_0.Pos
bits_0.Val = bit.Val + bits_1.Val
bits_0.Len = bits_1.Len + 1
One function per nonterminal

\[
\begin{align*}
\text{evalRest} &:: \text{Rest} \rightarrow \text{Float} \\
\text{evalRest} \ \text{None} &\equiv 0 \\
\text{evalRest} (\text{Places} \ \text{bits}) &\equiv \text{val} \\
\text{where} \\
&\quad (_\text{len}, \text{val}) = \text{evalBits} \ \text{bits} \ \text{pos} \\
\text{pos} &\equiv -1
\end{align*}
\]
Mapping attribute grammars to Prolog with constraints
Mapping attribute grammars to Prolog with constraints

- Nonterminals become predicate.
- One clause per production.
- Attributes become logical variables.
- One position per attribute declaration.
- Use logical variables to cover more than L-attributed grammars.
- Use constraint-logic programming to cover more AGs.
A definite clause grammar (DCG) for BNF

number -> bits, rest.
bits -> bit.
bits -> bit, bits.
bit -> ['0'].
bit -> ['1'].
rest -> [].
rest -> ['.'], bits.

Grammar-aware formalism
related Definite Clause
Programs of (Prolog) which
can be translated to Prolog.
The corresponding Prolog program obtained by expansion of the DCG

\[
\text{number}(A, C) :- \text{bits}(A, B), \text{rest}(B, C).
\]

\[
\text{bits}(A, B) :- \text{bit}(A, B).
\]

\[
\text{rest}(A, A).
\]

\[
\text{rest}([\cdot | A], B) :- \text{bits}(A, B).
\]

\[
\text{bit}([\cdot'0' | A], A).
\]

\[
\text{bit}([\cdot'1' | A], A).
\]

Thus, an accumulator deals with parsing. 1st arg for initial input; 2nd arg for remaining input.
One predicate per nonterminal

\[
\text{number}(\text{Val0}) \rightarrow \\
\{ \text{Pos1 is Len1 - 1} \}, \\
\text{bits}(\text{Pos1, Len1, Val1}), \\
\text{rest}(\text{Val2}), \\
\{ \text{Val0 is Val1 + Val2} \}.
\]

ERROR: \text{is/2}: Arguments are not sufficiently instantiated

\[
\begin{align*}
\text{number} & \ ::= \ \text{bits} \ \text{rest} \\
\text{bits}.\text{Pos} & = \text{bits}.\text{Len} - 1 \\
\text{number}.\text{Val} & = \text{bits}.\text{Val} + \text{rest}.\text{Val}
\end{align*}
\]

Computation over attributes

"
One predicate per nonterminal

number(Val0) →
  bits(Pos1, Len1, Val1),
  \{ Pos1 is Len1 - 1 \},
  rest(Val2),
  \{ Val0 is Val1 + Val2 \}.

Same error as before while trying “2 ^ bit.Pos”

number ::= bits rest

bits.Pos = bits.Len - 1
number.Val = bits.Val + rest.Val
One predicate per nonterminal
(Verwendung von CLP(R))

\[
\text{number(Val0) \rightarrow bits(Pos1, Len1, Val1), rest(Val2),}\]
\[
\{ \{ \text{Pos1} =: \text{Len1} - 1 \} \},\]
\[
\{ \{ \text{Val0} =: \text{Val1} + \text{Val2} \} \}.\]

Computation over attributes

\[
\text{number ::= bits rest}\]
\[
\text{bits.Pos} = \text{bits.Len} - 1\]
\[
\text{number.Val} = \text{bits.Val} + \text{rest.Val}\]

Constraint form
One predicate per nonterminal
(Leverage CLP(R))

\[
\text{bit}(_{\text{Pos}}, \text{Val}) \rightarrow \\
[\text{'0'}], \\
\{ \{ \text{Val} =:= 0 \} \}.
\]

\[
\text{bit}(_{\text{Pos}}, \text{Val}) \rightarrow \\
[\text{'1'}], \\
\{ \{ \text{Val} =:= 2 ^ {\text{Pos}} \} \}.
\]
One predicate per nonterminal

(Leverage CLP(R))

\[
\begin{align*}
\text{bits}(\text{Pos}, \text{Len}, \text{Val}) & \rightarrow \\
& \text{bit}(\text{Pos}, \text{Val}), \\
& \{ \{ \text{Len} = := 1 \} \}.
\end{align*}
\]

\[
\begin{align*}
\text{bits}(\text{Pos}_0, \text{Len}_0, \text{Val}_0) & \rightarrow \\
& \text{bit}(\text{Pos}_0, \text{Val}_1), \\
& \text{bits}(\text{Pos}_1, \text{Len}_1, \text{Val}_2), \\
& \{ \{ \text{Pos}_1 = := \text{Pos}_0 - 1 \} \}, \\
& \{ \{ \text{Len}_0 = := \text{Len}_1 + 1 \} \}, \\
& \{ \{ \text{Val}_0 = := \text{Val}_1 + \text{Val}_2 \} \}.
\end{align*}
\]
One predicate per nonterminal
(Leverage CLP(R))

rest(Val) -->
\[ \{ \{ \text{Val} =:= 0 \} \} \].

rest(Val) -->
\['.',]
bits(Pos, _Len, Val),
\[ \{ \{ \text{Pos} =:= -1 \} \} \].
The RepMin problem

program achieves its goal, since at first sight it seems that it is impossible to compute anything with this program. We will use this problem, and a sequence of different solutions, to build up understanding of a whole class of such programs.

In listing 1 we present the data type of interest, i.e. a `Tree`, which in this case stands for simple binary trees, together with their associated signature. The carrier type of an algebra is that type describing the objects of the algebra. We represent it by a type parameter to the signature type `Tree_Algebra`:

```haskell
type Tree_Algebra a = (Int -> a, a -> a -> a)
```

The associated evaluation function `cata_Tree` systematically replaces the constructors `Leaf` and `Bin` by their corresponding operations from the algebra that is passed as an argument.

We now want to construct a function `repMin :: Tree -> Tree` that returns a `Tree` with the same “shape” as its argument `Tree`, but with values at leaves replaced by the minimal value occurring in the original tree. In figure 1 an example of an argument with its result is given.

![Diagram of tree transformation](http://dx.doi.org/10.1007/10704973_4)
Computing the minimum

Designing and Implementing Combinator Languages

min_alg = (id, min::(Int->Int->Int))

replace_min :: Tree -> Tree
replace_min t = cata_Tree rep_alg t
where m = cata_Tree min_alg t

rep_alg = (const (Leaf m), Bin)

Listing 2: rm.sol1

Straightforward Solution

The straightforward solution to the Rep Min problem consists of a function in which cata Tree is called twice: once for computing the minimal leaf value, and once for constructing the resulting Tree. The function replace_min that solves the problem in this way is given in listing 2. Notice that the variable m is used as a global variable in the rep algebra, that in its turn is an argument to the tree constructing call of cata Tree. In figure 2 we have shown the flow of the data in a recursive call of cata Tree, when computing the minimal value. One of the disadvantages of this solution is that, since Bin Leaf min 3 4 Bin Leaf min 4 5 Leaf 3 3 4 4 5 5 Leaf 3 3 4 4 5 5

Fig. 2. Computing the minimum value

we call cata Tree twice, in the course of the computation the pattern matching associated with the inspection of the tree nodes is performed twice for each node in the tree. Although this is not a real problem in this solution we will try to construct a solution that calls cata Tree only once. We will do so by transforming the current program in a number of steps.

http://dx.doi.org/10.1007/10704973_4, “Designing and Implementing Combinator Languages” by S. Doaitse Swierstra, Pablo R. Azero Alcocer, João Saraiva
Building the result

Fig. 3. The flow of information when building the result

Fig. 4. The building blocks

Fig. 5. Tupling the computations

[http://dx.doi.org/10.1007/10704973_4, “Designing and Implementing Combinator Languages” by S. Doaitse Swierstra, Pablo R. Azero Alcocer, João Saraiva]
The abstract syntax of tree

[Leaf] Tree ::= Int

[Bin] Tree ::= Tree Tree
An attribute grammar for RepMin

[Start]  RepMin ::= Tree
          RepMin.result = Tree.result
          Tree.new = Tree.min

[Leaf]   Tree ::= Int
          Tree.result = makeLeaf(Tree.new)
          Tree.min = Int.val

[Bin]    Tree₀ ::= Tree₁ Tree₂
          Tree₀.min = min(Tree₁.min, Tree₂.min)
          Tree₁.new = Tree₀.new
          Tree₂.new = Tree₀.new
          Tree₀.result = makeBin(Tree₁.result, Tree₂.result)

An extra start symbol to distribute global minimum for replacement
The two cases for building the result
(Think of it as a visualized attribute grammar!)

Designing and Implementing Combinator Languages

Fig. 3. The flow of information when building the result

Fig. 4. The building blocks

Tupling the computations
The RepMin problem in Haskell
Abstract syntax in Haskell

```
data Tree = Leaf Int | Bin Tree Tree

Bin (Leaf 3) (Bin (Leaf 4) (Leaf 5))
```
**Use of two traversals**

*(This is not a direct mapping of the attribute grammar.)*

\[
\text{minTrav} :: \text{Tree} \rightarrow \text{Int} \\
\text{minTrav} (\text{Leaf } x) = x \\
\text{minTrav} (\text{Bin } l \ r) = \min (\text{minTrav } l) (\text{minTrav } r)
\]

\[
\text{repTrav} :: \text{Int} \rightarrow \text{Tree} \rightarrow \text{Tree} \\
\text{repTrav} x (\text{Leaf } _) = \text{Leaf } x \\
\text{repTrav} x (\text{Bin } l \ r) = \text{Bin } (\text{repTrav } x \ l) (\text{repTrav } x \ r)
\]

\[
\text{repMin} :: \text{Tree} \rightarrow \text{Tree} \\
\text{repMin } t = t' \\
\text{where} \\
\quad m = \text{minTrav } t \\
\quad t' = \text{repTrav } m \ t
\]

**Challenges**

1) Can we define an attribute grammar instead?
2) Can we use the attribute grammar to perform just one traversal?
Direct mapping of the AG to Haskell: one traversal (while requiring laziness)

\begin{verbatim}
repMinTrav :: Tree -> Int -> (Int, Tree)

repMinTrav (Leaf x) new
  = (x, Leaf new)

repMinTrav (Bin l r) new
  = (min ml mr, Bin tl tr)
where
  (ml, tl) = repMinTrav l new
  (mr, tr) = repMinTrav r new

repMin :: Tree -> Tree
repMin t = t'
where
  (m, t') = repMinTrav t m
\end{verbatim}

\begin{verbatim}
data Tree
  = Leaf Int
  | Bin Tree Tree
\end{verbatim}
Mapping attribute grammars to Prolog with unification
Mapping rules for Prolog

- One parameter position for input pattern.
- One parameter position for tree-typed result.
- Use logical variables for unknown attributes.
- Use unification to propagate eventual attribute values.
- No CLP needed because no operations needed on unknowns.
repMin(T1, T2) :-
    tree(New, T1, Min, T2),
    New = Min.

New and Min could be just one variable. We defer unification just for clarity.

```
[Start]  RepMin ::= Tree
         RepMin.result = Tree.result
         Tree.new = Tree.min

[Leaf]   Tree ::= Int
         Tree.result = makeLeaf(Tree.new)
         Tree.min = Int.val

[Bin]    Tree0 ::= Tree1 Tree2
         Tree0.min = min(Tree1.min, Tree2.min)
         Tree1.new = Tree0.new
         Tree2.new = Tree0.new
         Tree0.result = makeBin(Tree1.result, Tree2.result)
```
Concluding remarks

Attribute grammars support language implementation:

- analyses, translations, ...

By default, they are highly declarative.

They can be implemented in declarative languages.

The basic formalism lacks some expressiveness.

One might want to look at an advanced system (JastAdd).