3 Attribute Grammars

CFG’s, though capturing many of the syntactic as well as some semantic constraints in programming languages. They still lack the power to express all of them. For example, in many languages, variables used in program statements must be declared somewhere, and in some, must be declared prior to their use. This is typical of a class of restrictions found in programming languages. To give formal expression to such restrictions we might try travelling down the Chomsky Hierarchy where there is certainly an adequate grammar type. However embedding restrictions, natural to programming languages, in a Chomsky CS (type 0) grammars is both awkward and difficult to parse efficiently. These restrictions are not naturally expressable using these grammars. A more natural way to specify these constraints is to choose a set of variables, called attributes. Associated with each node in a parse tree are a set of attribute values. Such values can be constants or functions (including conditionals) of the attribute values at children’s or parent, or siblings nodes. That implies that attribute functions and values at a node can be associated with rules of the grammar. Each rule having its collection of attribute functions and values. A special conditional, can be used to validate or invalidate a string which, though parse-able by the given grammar, must also satisfy the attribute condition.

So an Attribute grammar consists of
1. A CFG, G.
2. A finite set of attributes, A.
3. A finite set of attribute identifiers, I, of the form \( A(N) \) or \( N_A \), where \( A \) is an attribute and \( N \) an NTS in G.
4. A finite set of conditions, \( ==, >, \) etc.
5. For each rule in the CFG, a finite set of assignment statement of the form
   \[ \langle \text{attribute identifier} \rangle \leftarrow \langle \text{exp} \rangle, \] in which \( \langle \text{exp} \rangle \) is a function on the attribute identifiers. \( \langle \text{exp} \rangle \) can involve conditionals, \( , \) and any of a set of operators, ex., \( +, * \) etc., as well as intermediate variables. There is at most one assignment per attribute.
6. In the assignment to an attribute identifier of form \( A(N_0) \leftarrow f(A(N_1), \ldots, A(N_n)) \) (each A can be any of the attributes in the attribute grammar)
   a) \( N_0 \) is from the rule leftside, \( (N_1, \ldots, N_n \) being from the rule rightside) then \( A(N_0) \)
      then \( A \) is is synthesized. i.e., the attribute of node \( n \) depends on those of its children.
   b) \( N_0 \) is from the rule rightside , \( (N_1, \ldots, N_n \) being from the rule left and/or rightside)
      then attribute \( A \) is inherited. In other words the attribute of node \( n \) depends on the attributes of \( n \)'s parent and perhaps siblings. But dependence on sibling can be passed to the parent and thus could be eliminated.
7. There is a special conditional expression: \( \text{condok} : \langle \text{boolean exp} \rangle \) in which must be true for a string to be parsed must satisfy in addition to being parse-able in \( G \)

Sometimes we allow also the conditional
\[
\text{if } ( \text{boolean}(A(N_1), \ldots, A(N_n)) ) A(N_0) \leftarrow f(A(N_1), \ldots, A(N_n));
\]
as well as other programming functions on the attributes of the right and left side of a rule

In the following examples NTS subscripts give the position of that NTS counting from its leftmost appearance (count. In a number of cases the same example is given in two different commonly used notations.

Examples:
An Attribute Grammar For $a^i b^j c^i$

(pure synthesis)

\[ S \rightarrow X \quad \text{condok : } \text{cnt}(X) = 0 \]
\[ X \rightarrow aXc \quad \text{cnt}(X_1) \leftarrow \text{cnt}(X_2) - 1 \]
\[ X \rightarrow Y \quad \text{cnt}(X) \leftarrow \text{cnt}(Y) \]
\[ Y \rightarrow bY \quad \text{cnt}(Y_1) \leftarrow \text{cnt}(Y_2) + 1 \]
\[ Y \rightarrow b \quad \text{cnt}(Y) \leftarrow 1 \]

An Attribute Grammar For $a^i b^j c^i$

(pure inheritance)

\[ S \rightarrow X \quad \text{cnt}(X) \leftarrow 0 \]
\[ X \rightarrow aXc \quad \text{cnt}(X_2) \leftarrow \text{cnt}(X_1) + 1 \]
\[ X \rightarrow Y \quad \text{cnt}(X) \leftarrow \text{cnt}(Y) \]
\[ Y \rightarrow bY \quad \text{cnt}(Y_2) \leftarrow \text{cnt}(Y_1) - 1 \]
\[ Y \rightarrow b \quad \text{condok cnt}(Y) = 1 \]

An Attribute Grammar For $a^i b^j c^i$

(inheritance and synthesis)

\[ S \rightarrow X \quad \text{cnt2}(X) \leftarrow 0 \]
\[ X \rightarrow aXc \quad \text{cnt2}(X_2) \leftarrow \text{cnt2}(X_1) + 1 \]
\[ X \rightarrow Y \quad \text{condok: cnt1}(Y) = \text{cnt2}(X) \]
\[ Y \rightarrow bY \quad \text{cnt}(Y_1) \leftarrow \text{cnt}(Y_2) + 1 \]
\[ Y \rightarrow b \quad \text{cnt1}(Y) \leftarrow 1 \]

An Attribute Grammar For $c^i$ [a and b are the attributes]

(inheritance and synthesis - undecidable!)

\[ S \rightarrow X \quad a(X) \leftarrow 0 \]
\[ b(X) \leftarrow 0 \]
\[ X \rightarrow cX \quad a(X_1) \leftarrow b(X_2) + 1 \]
\[ b(X_2) \leftarrow a(X_1) + 1 \quad \text{Deadly Enbrace} \]
\[ X \rightarrow c \quad \text{condok: a}(X) > 100 \]

Attribute Dependence Graph

An attribute grammars can yield indeterminate parses, i.e., parses to which it is impossible to make consistent attribute assignments. There is a test for a consistent assignment using an attribute dependency graph. Such a graph is constructed from a parse tree T, by placing the attribute assignments on the appropriate notes.
It should be noted that as long as we do not restrict the operation used in an attribute assignment statement we can always do anything that can be done with attributes using pure synthesized transmission of these attributes. This is so because the string being parsed can be passed to the root of the parse tree where any decision decideable can be made the condition for acceptance of that string. In general this is not a good idea because it necessitates the transmission of large amounts of data over considerable distance from the bottom to top of the parse tree, and then the execution of what is likely to be an excessively complex function at the root. It is better to have a number of simpler functions distributed over the parse tree.

Consider a few more simple examples:

**Identifier Of Length Less Than Or Equal To 4**

\[
\text{condok: } \text{cnt(id)} \leq 4
\]

\[
\begin{align*}
<\text{id}> & \rightarrow <\text{id}> \\
<\text{id}> & \rightarrow <\text{ltr}> <\text{id}> \\
<\text{ltr}> & \rightarrow \text{a | b |...| z}
\end{align*}
\]

**Binary Numbers With Value Greater Than 7 (RG, NTS on Left)**

\[
\text{condok: } \text{num(bin)} > 7
\]

\[
\begin{align*}
<\text{bin}> & \rightarrow <\text{bin}>0 \\
<\text{bin}> & \rightarrow <\text{bin}>1 \\
<\text{bin}> & \rightarrow 0 \\
<\text{bin}> & \rightarrow 1
\end{align*}
\]

Another example of: Binary Numbers With Value Greater Than 7 (RG, NTS on Right)
The grammar has 2 attributes, num which is the decimal value and np which is the bit position + 1.

\[
\begin{align*}
\text{condok: } & \text{num(bin)} > 7 \\
<\text{bin}> & \rightarrow <\text{bin}> \\
& \rightarrow 0<\text{bin}> \\
& \rightarrow 1<\text{bin}> \\
& \rightarrow 0 \\
& \rightarrow 1
\end{align*}
\]

More generally we would like a grammar in which every decorated parse tree is guaranteed to be acyclic, with a consistent ordering of the dependence graph. In the simple case of pure synthesis for example, this is true. It is also true for the grammar illustrated in figure 1, though the ordering necessary in general is much more complex to describe. If it is efficient to parse bottom-up/top-down for a given CFG then purely synthesized/inherited attributes can clearly also be handled efficiently during the parse. In many cases however when information is to be transmitted by attributes over large segments of program, as in checking declatation and types of variables, it is very inefficient to pass attributes purely in a single mode.
Pairs Of Identifiers
A string of letters (<id>) in which no letter is repeated followed by ":" then a second string of letters (<perm>) which only contain letters in the first letter string.

An example of a parse is given in figure 3 in which the conditional which is called when a terminal is acquired finds that:

Before the : a.{}, then b.{a}, then d.{ab}, c.{abd}, b.{abcd}, c.{abcd}, b.{abcd}.

Odd Number Of a's and Even number Of b's) In Any Order

The CFG by itself produces all sequences of as and bs. It is essentially a RG and so is easily parsed.

Example -(The RG Parse Tree-Decorated)
A Note On Attribute Grammar And Translation

An attribute grammar consists of a CFG, each of whose rules is accompanied with a set of attribute assignment functions. If the CFG is viewed as an input language and the value of one of the attributes is viewed as the output language, an attribute grammar can be used to formally describe translations. Instead of using a cond command at a print command to print the output language attribute(s) at strategic nodes. The print attribute is interpreted as the translation of the string parsed by the CFG part of the attribute grammar.
Here is an example in which an attribute grammar for translating from a prefix algebraic expression to a properly parenthesized infix expression by evaluation of the attributes.

\[
S \rightarrow B \\
B \rightarrow v \\
B \rightarrow x \ B \ B \\
\text{if } t(B_2) = x \& t(B_3) = x \text{ then } s(B_1) \rightarrow s(B_2) \times s(B_3); \quad t(B_1) \rightarrow x \\
\text{if } t(B_2) = + \& t(B_3) = x \text{ then } s(B_1) \rightarrow (s(B_2)) \times s(B_3); \quad t(B_1) \rightarrow x \\
\text{if } t(B_2) = x \& t(B_3) = + \text{ then } s(B_1) \rightarrow s(B_2) \times (s(B_3)); \quad t(B_1) \rightarrow x \\
\text{if } t(B_2) = + \& t(B_3) = + \text{ then } s(B_1) \rightarrow (s(B_2)) \times (s(B_3)); \quad t(B_1) \rightarrow x \\
B \rightarrow + \ B \ B
\]

Here is another example. The input string, s, is an algebraic expressions involving the variable v, and the operators * and + with * having precedence over +. The translation t(s) produces a sequence of assembly level 3-address instructions which give the same result as s.

\[
S \rightarrow E \\
E \rightarrow F + E \\
E \rightarrow F \\
F \rightarrow v * F \\
F \rightarrow v
\]

\[
s(E) \leftarrow 1, \quad \text{print : } t(E) \\
t(E_1) \leftarrow t(F); \quad t(E_2); \quad Is(E) := Is(F) + Is(E_2), \\
s(E_2) \leftarrow s(E_1)1, \quad s(F) \leftarrow s(E_1)0 \\
t(E) \leftarrow t(F), \quad s(F) = s(E) \\
t(F) : t(F_1) \leftarrow l(F_1) := v \times l(F_2), \\
s(F_2) \leftarrow s(F_1)0 \\
i(F) \leftarrow v
\]

\(t(B_j)\) is the operator, +, or x last done in Bj. If it was + and Bj generates a string whose last operation was x then the string that Bj translates to, s(B_j), should be parenthesized.
\[
E \rightarrow \text{let } U = Z \text{ in } Y \\
U \rightarrow X \\
Z \rightarrow R \\
R \rightarrow X \\
Y \rightarrow T \\
T \rightarrow X \\
X \rightarrow a^1 a \\
\]

**Attributes**

- \( E \rightarrow \text{eval} U_v = Z_v \) in \( Y_v \)
- \( U_v = \text{var}(X) \)
- \( Z_v = R_v \)
- \( R_v = X_v \)
- \( Y_v = T_v \)
- \( T_v = X_v \)
- \( X_v = a^1 a \)
- \( X_v = b^1 b \)
- \( X_v = c^1 c \)

\( x = \text{the variable } x, \ x^1 = \text{value of}(x) \)
<table>
<thead>
<tr>
<th>GRAMMAR</th>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>E ---&gt; let U = Z in Y</td>
<td>( U_v = Z_v ) in ( Y_v ), ( Y_v1 = U_v ) ( E_v = \text{eval}(Z_v) )</td>
</tr>
<tr>
<td>U --&gt; X</td>
<td>( U_v = \text{var}(X) )</td>
</tr>
<tr>
<td>( Z^1 ) ---&gt; ( Z^2 + R )</td>
<td>( Z_v^1 = Z_v^2 + R_v )</td>
</tr>
<tr>
<td>( Z ) ---&gt; ( R )</td>
<td>( Z_v = R_v )</td>
</tr>
<tr>
<td>( R^1 ) ---&gt; ( R^2 \star X )</td>
<td>( R_v^1 = R_v^2 \star X_v )</td>
</tr>
<tr>
<td>( R ) ---&gt; ( X )</td>
<td>( R_v = \text{val}(X_v) )</td>
</tr>
<tr>
<td>( Y^1 ) ---&gt; ( Y^2 + T )</td>
<td>( Y_v^1 = Y_v^2 + T_v ) ( Y_v1^2 = Y_v1^1 ), ( T_v1 = Y_v1 )</td>
</tr>
<tr>
<td>( Y ) ---&gt; ( T )</td>
<td>( Y_v = T_v ) ( T_v1 = Y_v1 )</td>
</tr>
<tr>
<td>( T^1 ) ---&gt; ( T^2 \star X )</td>
<td>( T_v^1 = T_v^2 \star X_v ) ( T_v1^2 = T_v1^1 ) ( X_v1 = T_v1 )</td>
</tr>
<tr>
<td>( T ) ---&gt; ( X )</td>
<td>( \text{if}(\text{var}(X_v) == X_v1) \text{then } T_v = X_v1 \text{ else } T_v = X_v )</td>
</tr>
<tr>
<td>( X ) --&gt; ( a^1 ) a</td>
<td>( X_v = a^1 a ) ( x = \text{the variable } x ),</td>
</tr>
<tr>
<td>( b^1 ) b</td>
<td>( X_v = b^1 b ) ( x^1 = \text{value of}(x) )</td>
</tr>
<tr>
<td>( c^1 ) c</td>
<td>( X_v = c^1 c )</td>
</tr>
</tbody>
</table>

x = the variable x, \( x^1 = \text{value of}(x) \)
Consider the left Regular grammar, G:

1. S ---> a  \quad \text{print}(1)
2. X ---> b  \quad \text{print}(2)
3. S ---> aS  \quad \text{print}(3)
4. S ---> aX  \quad \text{print}(4)

Here is DB representing the rules for the grammar:

1. s( [a], [1] ).
2. x( [b], [2] ).
3. s( [a| L1], [3 | R] ) :- length(L1, N), N > 0, s(L1, R).
4. s( [a| L1], [4 | R] ) :- length(L1, N), N > 0, x(L1, R).

Here is a query which, given a string generated by the grammar G, gives a sequence (list) of rules which generated that string. Following the query is a trace that shows the result and how it is developed.

?- s([a, a, b], R= [3, 4, 2]).\text{success}  \text{(Returned values for R are green)}
   1 s( [a], [1] ). fail
   2 x( [b], [2] ). fail
   3 s( [a| L1], [3 | R= [4,2] ) :- length([a, b], 2), 2 > 0,
      s([a, b], R). success
      1 s( [a], [1] ). fail
      2 x( [b], [2] ). fail
   3 s( [a| L1], [3 | R] ) :- length([ b ], 1), 1 > 0,
      \text{s([b], R). All Fail Backup}
      1 s( [a], [1] ). fail
      2 x( [b], [2] ). fail
      3 s( [a| L1], [3 | R] ) :- length(L1, N), N > 0, s(L1, R).fail
      4 s( [a| L1], [4 | R] ) :- length(L1, N), N > 0, x(L1, R).fail
      \text{All Fail Backup}
   4 s( [a| L1], [4 | R=2] ) :- length(L1, 1), 1 > 0,
      x(L1, R= [2] )\text{.success}
      1 s( [a], [1] ). fail
      2 s( [b], [2] ). success

-------------Next Page----------------------->

Programing Assignment 4 Prolog Assigned Nov 24--Due Dec 12.