CONTENTS

2 ATTRIBUTE GRAMMARS Definitions

3 Examples Of ATTRIBUTE GRAMMARS Synthesis, Inheritance
4 More Examples Of ATTRIBUTE GRAMMARS Binary-Decimal
4 Attribute Dependence Graph (Is The Parse + Attribute Decoration Viable?)

5 Attribute and Regular Grammars to Accept Even Number of as and an Odd Number of bs
5 Translating: Prefix Algebraic Expression to a Properly Parenthesized Infix Expression

6 A Note On Attribute Grammar And Translation (Translational Semantics)
6 Translation From Algebraic Expression To Sequence of Generic Machine Operations
7 Any String Of Letters, S1, Followed By Second Sting Which Only Contain Letters in S1

8 SUMMARY-LEX: FSA-RG-RE, GEN PARSE: CFG-ATT-G, TRANSLATION ATT-G

© 2006 M C Paull
3 Attribute Grammars

CFG’s, though capable of specifying most syntactic, (and some semantic) properties of a programming language, nevertheless lack sufficient power to express certain important non-local syntactic relations in a such languages. 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 Context Sensitive (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, 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 a Pure Synthesis-Attribute Grammar consists of (See example Below)

I. A CFG, G, Describing Language, L(G) and
A set of attributes, AG associated with G and

Associated with each rule, r, of G,

r: N0 --> β where β is a string of TSs and NTSs designated N_i i = 1 to n_r
with i increasing from left to right in β. In addition one or more

Attribute Statements of the form
N_0 <-- f(a_1(N_1), ...a_n(N_n)), or
if ( boolean(a_1(N_1), ...,a_n(N_n)) a_1(N_0) <-- f(A(N_1), ...,A(N_n)) or
condok: (boolean(a_1(N_1), ...,a_n(N_n)));

where a_i e AG ,

f is a function - virtually any function-
boolean is a boolean function, on its arguments, ex. &&,++ .
The condok attribute statement is associate with a rule of the grammar
so that when that rule is used in a parse the boolean function associated
with it can be evaluated-iff it is true the parse is acceptable and the string
parsed is in the language

An Attribute Grammar For Language: \{ a^i b^j c^l | i =1,2, .... \} ex. \{ a^3 b^3 c^3 \} = aaabbbccc

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Attribute Statements</th>
<th>Attribute: = cnt(NTS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S ---&gt; X</td>
<td>condok : cnt(X) &lt;-- 0</td>
<td></td>
</tr>
<tr>
<td>X ---&gt; aXc</td>
<td>cnt(X_1) &lt;-- cnt(X_2) - 1</td>
<td></td>
</tr>
<tr>
<td>X ---&gt; Y</td>
<td>cnt(X) &lt;-- cnt(Y)</td>
<td></td>
</tr>
<tr>
<td>Y ---&gt; bY</td>
<td>cnt(Y_1) &lt;-- cnt(Y_2) + 1</td>
<td></td>
</tr>
<tr>
<td>Y ---&gt; b</td>
<td>cnt(Y) &lt;-- 1</td>
<td></td>
</tr>
</tbody>
</table>

NTS subscripts (ex. in the example X_1) give the position of that NTS in the associated CFG
rule counting from its leftmost appearance therein.

Example

ATTRIBUTE GRAMMARS Definitions

© 2008 M C Paull 2
An Attribute Grammar For Language: \{ a^i b^j c^l \mid i = 1,2, \ldots \}

**Attributes:**
- \texttt{cnt}(NTS)

**Rule 1:**
- \texttt{S} \rightarrow \texttt{X}
- condok: \texttt{cnt}(\texttt{X}) == 0
  - \texttt{cnt}(\texttt{X}_1) \leftarrow \texttt{cnt}(\texttt{X}_2) - 1
  - \texttt{cnt}(\texttt{Y}) \leftarrow \texttt{cnt}(\texttt{Y} + 1)
  - \texttt{cnt}(\texttt{Y}_1) \leftarrow \texttt{cnt}(\texttt{Y}_2) + 1

**Rule 2:**
- \texttt{Y} \rightarrow \texttt{b}
  - \texttt{cnt}(\texttt{Y}) \leftarrow 1

**CFG + ATTRIBUTES AT RULES**

---

An Attribute Grammar For Language: \{ a^i b^j c^l \mid i = 1,2, \ldots \}

**Attributes:**
- \texttt{cnt1}(NTS), \texttt{cnt2}(NTS)

**Rule 1:**
- \texttt{S} \rightarrow \texttt{X}
  - \texttt{cnt}(\texttt{X}) \leftarrow 0
  - \texttt{cnt}(\texttt{X}_1) \leftarrow \texttt{cnt}(\texttt{X}_1) + 1

**Rule 2:**
- \texttt{Y} \rightarrow \texttt{b}
  - \texttt{cnt}(\texttt{Y}) \leftarrow \texttt{cnt}(\texttt{Y} - 1)

**Rule 3:**
- \texttt{Y} \rightarrow \texttt{b}
  - \texttt{cnt}(\texttt{Y}) \leftarrow 1

---

An Attribute Grammar For Language: \{ a^i b^j c^l \mid i = 1,2, \ldots \}

**Attributes:**
- \texttt{a}(NTS), \texttt{b}(NTS)

**Rule 1:**
- \texttt{S} \rightarrow \texttt{X}
  - \texttt{a}(\texttt{X}) \leftarrow 0
  - \texttt{b}(\texttt{X}) \leftarrow 0

**Rule 2:**
- \texttt{X} \rightarrow \texttt{cX}
  - \texttt{a}(\texttt{X}) \leftarrow \texttt{b}(\texttt{X}) + 1
  - \texttt{b}(\texttt{X}) \leftarrow \texttt{a}(\texttt{X}) + 1

---

An Attribute Grammar For Identifiers Of Length Less Than Or Equal To 4

**condok:** \texttt{cnt(id)} <= 4

**Examples Of ATTRIBUTE GRAMMARS**

© 2006 M C Paull
Another example of: Binary Numbers With Value Greater Than 7 (RG, NTS on Left)
The grammar has 2 attributes, num which is the decimal value and np which is the bit position + 1.

<table>
<thead>
<tr>
<th>&lt;okbin&gt;</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bin&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;bin&gt;</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>&lt;bin&gt;</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

condok: num(bin) > 7
num(bin0) ← 2 × num(bin1)
num(bin0) ← 2 × num(bin1) + 1
num(bin) ← 0
num(bin) ← 1

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 nodes.

An attribute dependency graph, D, has a vertex corresponding to each <node, attribute> pair, <n,a>, of T. Iff <n,a> of T depends on <node, attribute> pair, <n1,a1> of T, there is an edge in D from vertex <n1,a1> to vertex <n,a>. The arrows in figure 1 is such a graph. If there are no cycles (acyclic) in an attribute dependency graph, D, then the attributes can be consistently evaluated. A topological sort, orders the vertices of D in the order v1 then v2, then ... then vn-1 then vn with the guarantee that there is no edge in D from vj in this ordering if there is no edge from w to v in D, has edges from vertices the vertices in such gives the order in which a successful assignment of attribute values to nodes of the parse tree.

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 declaration and types of variables, it is very inefficient to pass attributes purely in a single mode.

The design of an Attribute grammar would start with a CFG which defines the language desired as closely as possible. Usually this will result in grammar for a language which includes what is wanted, but also includes strings which are not wanted in the language. Then attributes are invented with some direction of propagation through a parse tree which will check for such undesirable features.

© 2006 M C Paull
Translating from a Prefix Algebraic Expression to a Properly Parenthesized Infix Expression

\[ t(B_j) \text{ is the operator, } +, \text{ or } x \text{ last done in } B_j \text{. If it was } + \text{ and } B_j \text{ generates a string whose last operation was } x \text{ then the string that } B_j \text{ translates to, } s(B_j), \text{ should be parenthesized,} \]

\[ L[AG] = \text{Odd Number of a's and Even number of b's} \text{) In Any Order} \]
A Note On Attribute Grammar And Translation (Translational Semantics)

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 another example. The input string $S_{\text{in}}$ is an algebraic expressions involving the variable $v$, and the operators * and + with * having precedence over +. The translation attribute $t(\ )$ produces a sequence of assembly level 3-address instructions which give the same result as $S_{\text{in}}$. Numbers travel down the parse tree changing as they go to supply the temporary addresses to which intermediate results are assigned. They do this with attribute $s(\ )$. Also the temporary addresses travel up sob to supply arguments for operations represented at ancestors with attribute $u(\ )$. The code produced at the a level of the tree are tagged unto the ends of the operations generated at the next highest level.

\[
\begin{align*}
S \rightarrow & E & s(E) = 1, \quad \text{print} : t(E) \\
E \rightarrow & F + E & s(F) = s(E_1)^\ast 0^\ast \quad t(E_1) \leftarrow \text{"T" \, } s(E_1) = u(F) + u(E_1) ; t(E_2) ; t(E) \quad u(E_1) = \text{"T" \, } s(E_1) \\
E \rightarrow & F & s(F) = s(E) \quad t(E) \leftarrow t(F), \quad u(F) = u(F) \\
F \rightarrow & v \ast F & s(F_2) = s(F_1)^\ast 1^\ast \quad t(F_1) \leftarrow \text{"T" \, } s(F_1) = v \ast t(F_2) ; \quad u(F) = \text{"T" \, } s(F_1) \\
F \rightarrow & v & t(F) \leftarrow \text{"T" \, } s(F) = v
\end{align*}
\]

Translation From Algebraic Expression To Sequence of Generic Machine Operations
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 next.

Any String Of Letters, S1, Followed By Second Sting Which Only Contain Letters in S1
**GENERAL PARSING 1** Natural Language Description
Structured Syntax of a Programming Language

Programming Language
**CFG** Description Grammar
Recognition-Minimum Structure
Parsing-Structure Necessary
(Ex. Pick out single operators and order them)
(Ex. Precedence Grammar)
Dangers for Parsing: Ambiguity

In general an $O(n^2)$ algorithmic necessary but for the subclass of interest is $O(n)$ ex LL(k).
If it is and this can be tested and the algorithm

**GENERAL PARSING 2** Natural Language Description
Very Structured Syntax of a Programming Language

Highly Structured Programming Language
**Attribute:** Grammar
CFG + Attributes on Rules

Parse CFG with Pushdown Automata $O(n)$
+ Tree Decoration $O(n)$ synthesis, inheritance

**LEXICOGRAPHIC UNIT** Natural Language Description
Structured (Much Less than for GENERAL PARSING)
Syntax of a Programming Language

If Deterministic algorithmic
If curably Non-Deterministic
Can use lookahead-Algorithmic
Algorithmic some algorithms may give NDRG but it can be transformed to a D RG

Parsing Algorithm Necessary
Finite State Automata $O(n)$
Input

Detect ids operators, punctuation
possible lookahead

Representation

Syntax for language L
many CFGs for L
$G_1, G_2, ..., G_{n-1}, G_n$

Only a Few Can be rapidly Parse and Reflect the Structure of L as the Appropriate CFGs must have these Properties

Input

Parse
Subtree

Stack

Un-Structured Language Description
RE - alphabet {1.,0}, | or *
operator=Kleene Closure

Dangers for Parsing (Not for pure Recognition):
Both signalled by Non-Determinism
Incurable Ambiguity
Curable Ambiguity by lookahead
For Recognition Deterministic and Non Deterministic Are OK-because any Non-Deterministic can be transformed to a Deterministic which Recognizes the same language