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

CONTENTS

© 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 diffcult 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, A_G associated with G and
Associated with each rule, r, of G,

r: N_0 **--->** 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_j e A_G ,

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: { $a^i b^i c^i$ | i = 1,2, ...} ex. { $a^3 b^3 c^3$ } = aaabbbccc

Grammar	(**pure synthesis**) **Attribute Staements** Attribute:= **cnt(** NTS)
S **---->** X	condok : **cnt**(X) <---- 0
X **---->** aXc	**cnt**(X_1) **<---- cnt**(X_2) - 1
X **---->** Y	**cnt**(X) **<---- cnt**(Y)
Y **---->** bY	**cnt**(Y_1) **<---- cnt**(Y_2) + 1
Y **---->** b	**cnt**(Y) **<----** 1

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

An Attribute Grammar For Language: { $a^i b^i c^i$ | i = 1,2, ...}

Attribute: **cnt**(NTS)

(**pure synthesis**)

CFG		ATTRIBUTES AT RULES
S → X		condok : **cnt**(X) == 0
X → aXc		**cnt**(X_1) ← cnt(X_2) - 1
X → Y		**cnt**(X) ← cnt(Y)
Y → bY		**cnt**(Y_1) ← cnt(Y_2) + 1
Y → b		**cnt**(Y) ← 1

CFG + ATTRIBUTES AT RULES

left depends on right

Decorated Trees

An Attribute Grammar For Language: { $a^i b^i c^i$ | i = 1,2, ...}

Attribute: **cnt**(NTS)

(**pure inheritance**)

S	X	cnt(X)	0
X	aXc	cnt(X_2)	cnt(X_1) + 1
X	Y	cnt(X)	cnt(Y)
Y	bY	cnt(Y_2)	cnt(Y_1) - 1
Y	b	condok cnt(Y) ==	1

Decorated Trees

An Attribute Grammar For Language: { $a^i b^i c^i$ | i = 1,2, ...}

Attributes: **cnt1**(NTS), **cnt2**(NTS)

(**inheritance** and **synthesis**)

S → X	cnt2(X) ←	0
X → aXc	cnt2(X_2) ←	cnt2(X_1) + 1
X → Y	condok: cnt1(Y) ==	cnt2(X)
Y → bY	cnt(Y_1) ←	cnt(Y_2) + 1
Y → b	cnt1(Y) ←	1

right depends on left & left depends on right

cnt2(X) =1!= cnt1(Y) =2 cond NG

Decorated Trees

An Attribute Grammar For Language: ?????

Attributes: **a**(NTS) **b**(NTS)

(inheritence and synthesis - undecideable!)

S → X	a(X) ←	0
	b(X) ←	0
X → cX	a(X_1)	b(X_2) + 1
	b(X_2)	a(X_1) + 1
X → c	a(X) <----	0
	b(X) <----	0

Deadly Embrace

cycle

a(X_1)<---b(X_2)+1

b(X_2)<---a(X_1)+1

An AttributeGrammar Fo Identifiers Of Length Less Than Or Equal To 4

<okid>----><id>	**condok:** cnt(id) <= 4
<id> ----><ltr><id>	cnt(id_1)<---- cnt(ltr) + cnt(id_2)
<id> ----><ltr>	cnt(id) <---- cnt(ltr)
<ltr> ----> a | b |...| z	cnt(ltr) <----1

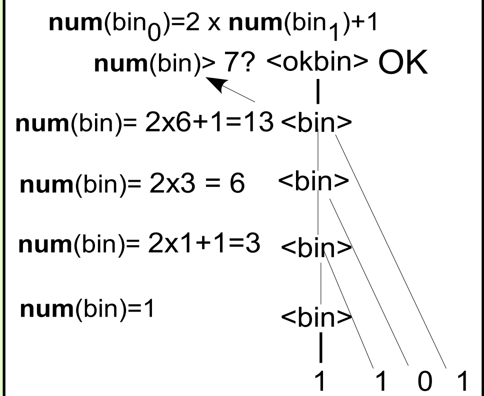
<okid>4 <= 4OK

Examples Of ATTRIBUTE GRAMMARS

© 2006 M C Paull

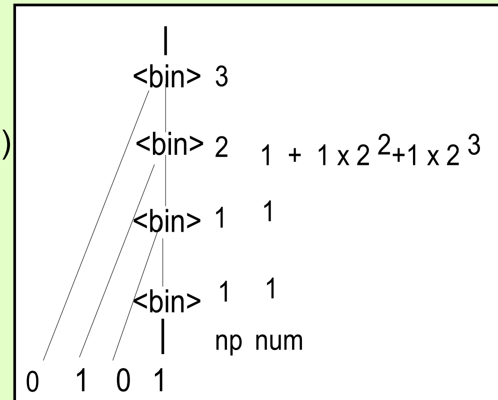
Binary Numbers With Value Greater Than 7 (RG, NTS on Left)

Production	Attribute rule
<okbin> → <bin>	**condok:** num(bin) > 7?
<bin> → <bin>0	**num**(bin_0) ← 2 x **num**(bin_1)
<bin> → <bin>1	**num**(bin_0) ← 2 x **num**(bin_1) + 1
<bin> → 0	**num**(bin) ← 0
<bin> → 1	**num**(bin) ← 1



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.

Production	Attribute rule
<okbin> → <bin>	**condok:** num(bin) > 7
<bin> → 0<bin>	num(bin_1) ← num(bin_2)
	np(bin_1) ← np(bin_2) + 1
<bin> → 1<bin>	num(bin_1) ← num(bin_2) + $2^{np(bin_2)}$
	np(bin_1) ← np(bin_2) + 1
<bin> → 0	num(bin) ← 0)
	np(bin) ← 1
<bin> → 1	num(bin) ← 1)
	np(bin) ← 1



More Examples Of ATTRIBUTE GRAMMARS Binary-Decimal

Attribute Dependence Graph (Is The Parse + Attribute Decoration Viable?)

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 $v_1$then v_2,then ... then v_{n-1}then v_n with the guarantee that there is no edge in D from v_j 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 necesary 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 efficientlty 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.

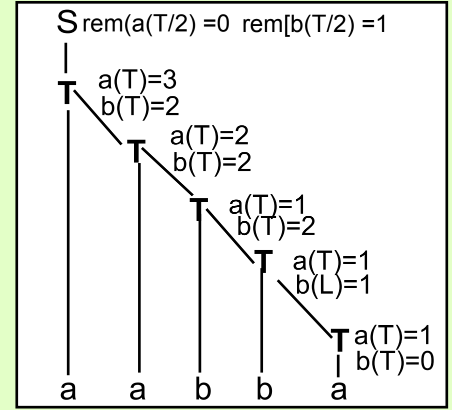
The design of an Attribute grammar would start with a CFG which defines the language desired as closely as possible. Usally this will result in grammar for a language which includes what is wanted, but also includes strings which are not wanted in the language. Then attribures are invented with some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

© 2006 M C Paull

Attribute Grammar (AG) Attribute a counts as
Attribute b counts bs

S---->T	condok: rem[a(T]/2)=1 & rem(b(T)/2)=0
T----> aT	$a(T_1)$ ----> $a(T_2)$ + 1 $b(T_1)$----> $b(T_2)$
T----> bT	$a(T_1)$ ----> $a(T_2)$ $b(T_1)$ ----> $b(T_2)$ +1
T----> a	a(T) ----> 1, b(T)--->0
T----> b	b(T) ----> 1, a(T)--->0

L[AG] = Odd Number of a' s and Even number of b' s) In Any Order



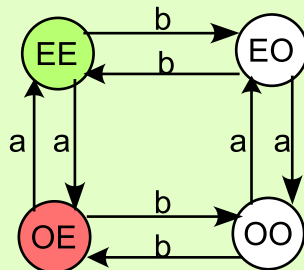
Regular Grammar (RG) equivalent FSA

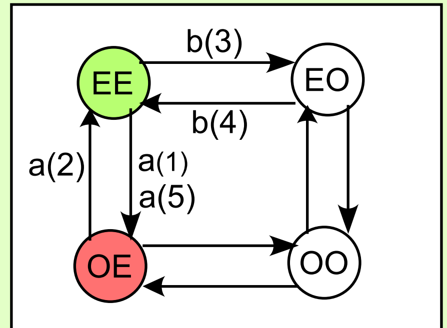
EE ----> aOE | bEO |a

EO ----> aOO | bEE

OE ----> aEE | bOO

OO ----> aEO | bOE |b

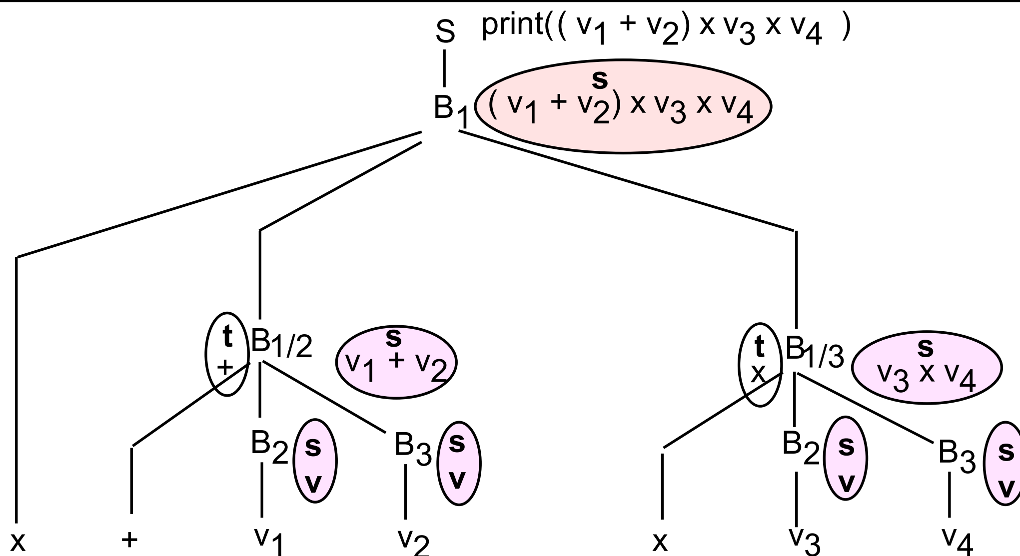




Attribute and Regular Grammars to Accept Even Number of as and an Odd Number of bs

AG

	Synthesized
S ⟶ B	**print** : s(B)
B ⟶ v	**s**(B) ⟶ v
B ⟶ x B B	**if t**(B_2) = x **&** t(B_3) = x **then** **s**(B_1) ⟶ **s**(B_2) x **s**(B_3); **t**(B_1) ⟶ x
	if t(B_2) = + **&** t(B_3) = x **then** **s**(B_1) ⟶ **(s**(B_2) **)** x **s**(B_3); **t**(B_1) ⟶ x
	if t(B_2) = x **&** t(B_3) = + **then** **s**(B_1) ⟶ **s**(B_2) x **(s**(B_3) **)**; **t**(B_1) ⟶ x
	if t(B_2) = + **&** t(B_3) = + **then** **s**(B_1) ⟶ **(s**(B_2) **)** x **(s**(B_3) **)**; **t**(B_1) ⟶ x
B ⟶ + B B	**s**(B_1) ⟶ **s**(B_2) + **s**(B_3); **t**(B_1) ⟶ +



t(B_j) is the operator, +, or x last done in B_j If it was + and B_j generates a string whose last operation was x then the string that B_j translates to, **s**(B_1), should be parenthesized,

Translating from a Prefix Algebraic Expression to a Properly Parenthesized Infix Expression

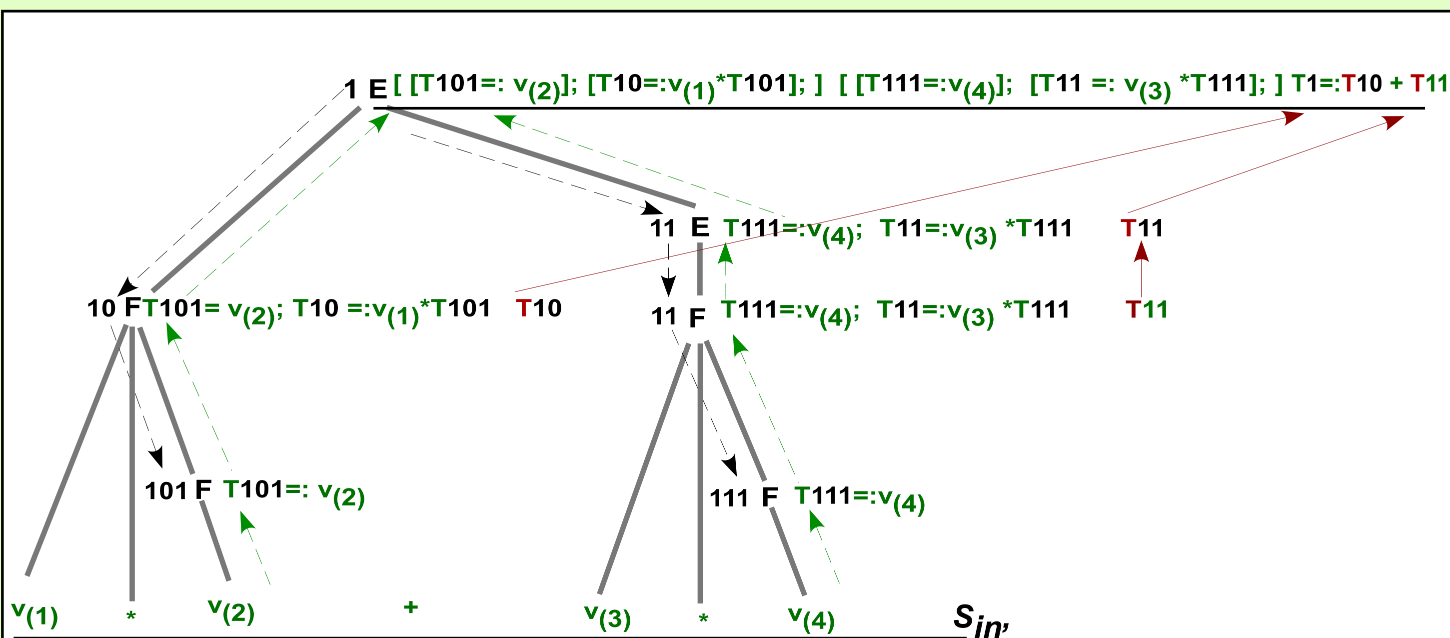
© 2006 M C Paull

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_{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_{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 producedat the a level of the tree are tagged unto the ends of the operations generated at the next highest level.

Rule	s	t	u
S----> E	s(E) =1,	print : t(E)	
E---->F + E	s(F) = s(E_1)"0" s(E_2) = s(E_1)"1"	t(E_1)<--- "T" s(E_1)= u(F) + u(E_2) ; t(F) ;t (E_2)	u(E_1)= "T" s(E_1)
E----> F	s(F) = s(E)	t(E) <--- t(F) ,	u(F) = u(F)
F----> v * F	s(F_2) = s(F_1)"1"	t(F_1)<--- t(F_2); "T"s(F_1) =: v * t(F_2) ;	u(F) = "T" s(F_1)
F---->v		t(F) <--- "T"s(F)=:v	



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.

Production	Attribute rules
<pair> → <okid>:<perm>	chkeq(perm) ← sv(okid)
<okid> → <id>	chk(id) ← {}
	sv(okid) ← sv(id)
<id> → <ltr><id>	chk(ltr) ← chk(id_1)
	chk(id_2) ← let(ltr) U chk(id_1)
	sv(id_1) ← sv(id_2)
<id> → <ltr>	sv(id) ← let(ltr) U chk(id)
<ltr> → a	cond : a *not in* chk(ltr) : let(ltr) ← {a}
.	
<ltr> → z	condok : z *not in* chk(ltr) : let(ltr) ← {z}
<perm> → <pltr>	chkeq(pltr) ← chkeq(perm)
<perm> → <pltr><perm>	chkeq(pltr) ← chkeq($perm_1$)
	chkeq($perm_2$) ← chkeq($perm_1$)
< pltr> → a	cond :a E chkeq(pltr)
.	
< pltr> → z	cond :z E chkeq(pltr)<pair>

Any String Of Letters, S1, Followed By Second Sting Which Only Contain Letters in S1

© 2006 M C Paull

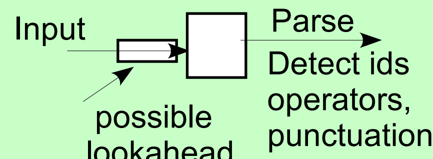
LEXICOGRAPHIC UNIT Natural Language Description
Structured [Much Less than for GENERAL PARSING)
Syntax of a Programming Language

If Deterministic-algorithmic
If curably Non Deterministic
Can use lookahead-Algorithmic

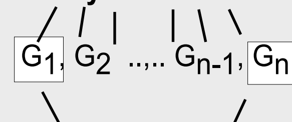
Programming Language
RG Description Grammar
X---->t, X---->tX
Recognition-Minimum Structure
Parsing-Structure Necessary
(EX. Pick out variable and operators)

Algorithmic some algorithms may give NDRG-but it can be transformed to a D RG

Parsing
Algorithm
Necessary
Finite State Automata O(n)



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



Representation

Un-Structed Language Description
RE - alphapet {1., 0), | or
* operator=Kleene Closure

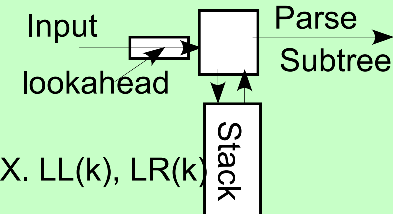
Syntax for language L
many CFGs for L
G_1, G_2 ..,.. G_{n-1}, G_n
Only a Few Can be rapidly Parse and Reflect the Structure of L as the Appropriate CFGs must have these Properties

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 outsingle operators and order them)
(EX. Precedence Grammar)
Dangers for Parsing: Ambiguity

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

Parsing
Algorithm
Necessary
Pushdown Automata O(n)



U-Structured Language Description
?

EX. LL(k), LR(k)

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, inheratance

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

© 2006 M C Paull