Attribute Grammars

Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de

June 11, 1997

– Wilhelm/Maurer: Compiler Design, Chapter 9 –

Attribute Grammars

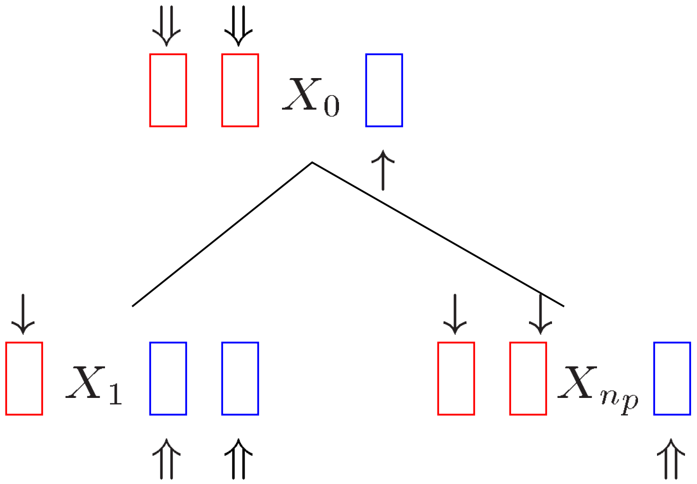
Attributes: containers for static semantic (non-context–free syntactic) information,

Directions: attributes

inherit information from the (upper) context,
synthesize information from information in subtrees,

Semantic rules: define computation of attribute values.

Attributes as Carriers of Context Information



Inherited

Synthesized

Example Grammar: Scoping

attribute grammar Scopes:
nonterminals $Stms,\ Stm,\ Decls,\ Decl,\ Id,\ Args,\ Ptype;$
domain $Env = String \to Types;$
attributes **syn** ok **with** $Decls,\ Decl,\ Stms,\ Stm$ **domain** $Bool;$
inh $e\text{-}env$ **with** $Stms,\ Stm,\ Decls,\ Decl$ **domain** $Env;$
inh $it\text{-}env$ **with** $Decls,\ Decl$ **domain** $Env;$
syn $st\text{-}env$ **with** $Decls,\ Decl$ **domain** $Env;$
syn $name$ **with** Id **domain** $String;$
syn $type$ **with** $Ptype,\ Args$ **domain** $Types;$

ok is true,

- if all used identifiers are declared, and
- if there are no multiple declarations of one identifier in the same scope.

it-env, st-env are "temporary environments", in which declarative information is collected.

e-env is the "effective" environment, in which procedure calls are type checked.

rules

$0: \quad Stms \quad \rightarrow \quad Stm$

$1: \quad Stms \quad \rightarrow \quad Stms\ ;\ Stm$

$Stms_0.ok = Stms_1.ok \text{ and } Stm.ok$

$2: \quad Stm \quad \rightarrow \quad \textbf{begin}\ Decls\ ;\ Stms\ \textbf{end}$

$Decls.it\text{-}env = \emptyset$

$Stms.e\text{-}env = Stm.e\text{-}env + Decls.st\text{-}env$

$Decls.e\text{-}env = Stm.e\text{-}env + Decls.st\text{-}env$

$Stm.ok = Decls.ok \text{ and } Stms.ok$

$3: \quad Decls \quad \rightarrow \quad Decl$

$4: \quad Decls \quad \rightarrow \quad Decls\ ;\ Decl$

$Decls_1.it\text{-}env = Decls_0.it\text{-}env$

$Decl.it\text{-}env = Decls_1.st\text{-}env$

$Decls_0.st\text{-}env = Decl.st\text{-}env$

$Decls_0.ok = Decls_1.ok \text{ and } Decl.ok$

$5: \quad Decl \quad \rightarrow \quad \textbf{proc}\ Id : Ptype\ \textbf{is}\ Stms$

$Decl.st\text{-}env = Decl.it\text{-}env + \{\ Id.name \mapsto Ptype.type\ \}$

$Stms.e\text{-}env = Decl.e\text{-}env$

$Decl.ok = undef(\ Id.name, Decl.it\text{-}env) \text{ and } Stms.ok$

$6: \quad Stm \quad \rightarrow \quad \textbf{call}\ Id\ (\ Args)$

$Stm.ok = def(Id.name, Stm.e\text{-}env) \text{ and}$
$\quad check(Args.type, Stm.e\text{-}env(Id.name))$

Attribute Grammars – Terminology

Let $G = (V_N, V_T, P, S)$ be a CFG, the *underlying* CFG. The p–th production in P is written as
$p : X_0 \;\rightarrow\; X_1 \ldots X_{n_p}$,
$X_i \in V_N \cup V_T$, $1 \leq i \leq n_p$, $X_0 \in V_N$.

An **attribute grammar (AG)** over G consists of

- two disjoint sets Inh and Syn of **inherited** resp. **synthesized** attributes,
- an association of two sets $Inh(X) \subseteq Inh$ and $Syn(X) \subseteq Syn$ with each symbol in $V_N \cup V_T$;
  - $Attr(X) = Inh(X) \cup Syn(X)$ set of all attributes of X;
  - $a \in Attr(X_i)$ has an **occurrence** in production p at occurrence X_i, written a_i.
  - $V(p)$ is the set of all attribute occurrences in production p.
- the association of a **domain** D_a with each attribute a;

- a **semantic rule**

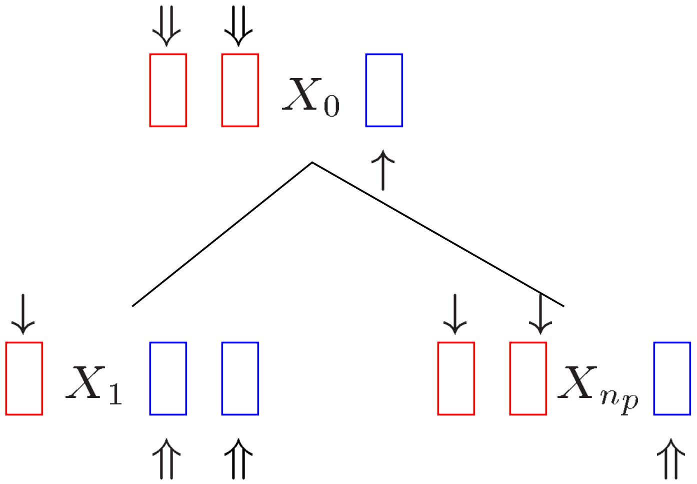
$$a_i = f_{p,a,i}\,(\,b^1_{j_1}, \ldots, b^k_{j_k}\,) \quad (0 \leq j_l \leq n_p)\;(1 \leq l \leq k)$$

for each **defining occurrence** of an attribute, i.e.,

– $a \in Inh(X_i)$ for $1 \leq i \leq n_p$ or
– $a \in Syn(X_0)$ in each production p,

where $b^l_{j_l} \in Attr(X_{j_l})$ $(0 \leq j_l \leq n_p)$ $(1 \leq l \leq k)$.
$f_{p,a,i}$ is thus a function from $D_{b^1} \times \ldots \times D_{b^k}$ to D_a.

Attributes as Carriers of Context Information



Inherited

Synthesized

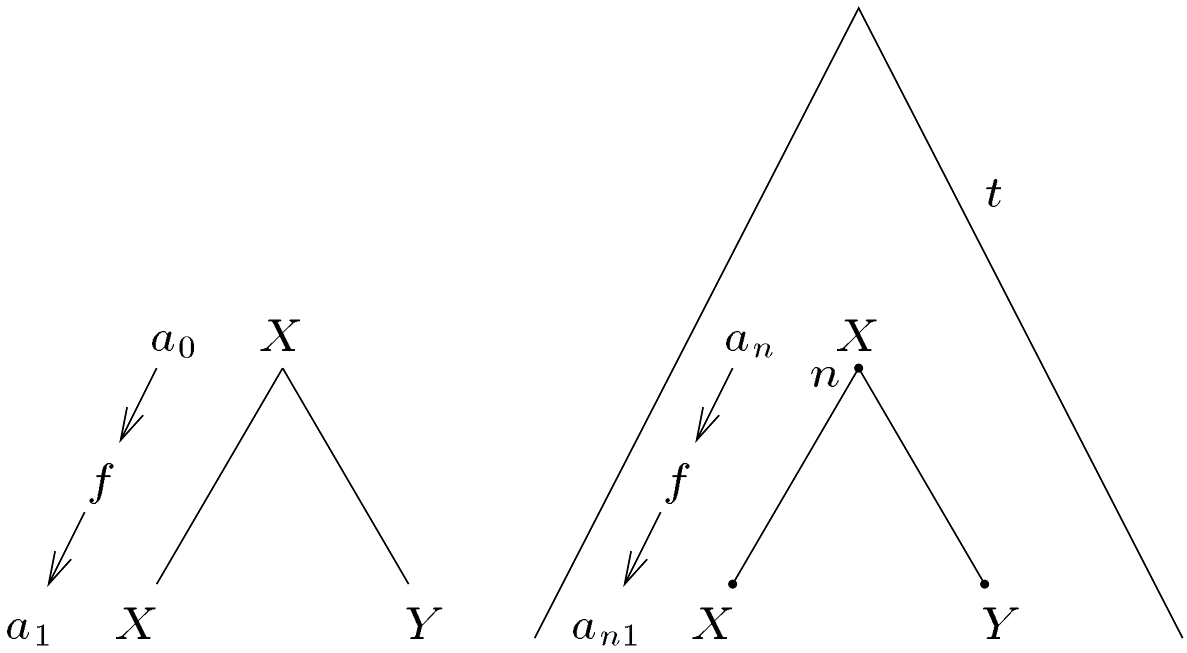
More Terminology

- Productions of the *underlying* CFG have **instances** in syntax trees.
- Node n labelled with $X \in V_N \cup V_T$ has an **instance** a_n of attribute $a \in Attr(X)$.
- Hence, there are

  attributes associated with non-terminals (and terminals),
  attribute occurrences in productions, and
  attribute instances at nodes of syntax trees.

- The semantic rule for a def. attribute occurrence in a production determines the values of all corresponding attribute instances in instances of the production.
- **Attribute Evaluation** is the process of computing the values of attribute instances in a tree using the semantic rules.

Attribute Occurrences and Attribute Instances



attribute occurrences
a_0, a_1

attribute instances
a_n, a_{n1}

A production and one of its instances

Semantics of an Attribute Grammar

Let t be a syntax tree to AG G,
$symb(n) \in V_N$,
$prod(n)$ be the production applied at n.

Attribute instance a_n of attribute $a \in Attr(symb(n))$ at n has to be given a value from D_a.

Semantic rule $a_i = f_{p,a,i}\,(\,b^1_{j_1}, \ldots, b^k_{j_k}\,)$ of $prod(n) = p$ induces the relation on the values of the attribute instances of the instance of $prod(n)$:

$$val(a_{ni}) = f_{p,a,i}\,(val(b^1_{nj_1}), \ldots, val(b^k_{nj_k}))$$

G induces a system of equations for t:

- variables are the attribute instances at the nodes of t,
- equations are defined by the above relation,
- recursion is not allowed, would not permit an evaluation of all attribute instances.

- AG, which never induces a recursive system of equations, is called **well formed**.

Normal Form

- Attribute occurrences a_i where
  - $a \in Inh(X_i)$ and $1 \leq i \leq n_p$ and
  - $a \in Syn(X_0)$

  are **defining occurrences**.

- All others are **applied occurrences**.

- AG is in **normal form**, if all arguments of semantic functions are applied occurrences.

Consequences of Normal Form:

- Semantic rules define values of def. occurrences in terms of appl. occurrences.

- Computation of the value of an attribute in one instance of a production (in a tree) requires the previous evaluation of an attribute in a neigbouring instance of a production.

- For later: Chains of attribute dependences inside a production have at most length one.

Short Circuit Evaluation of Boolean Expressions

The generated code:

- only load–instructions and conditional jumps;
- no instructions for **and, or** and **not**;
- subexpressions evaluated from left to right;
- for each (sub)expression, only the smallest subexpression is evaluated, which determines the value of the whole (sub)expression.

Code for the Boolean expression (a **and** b) **or** **not** c:

	LOAD a	
	JUMPF L1	jump-on-false
	LOAD b	
	JUMPT L2	jump-on-true
L1:	**LOAD** c	
	JUMPT L3	
L2:	Code for true–successor	
L3:	Code for false–successor	

Attribute grammar **BoolExp** describes

- code generation for short circuit evaluation,
- label generation for subexpressions,
- transport of labels for true– and false–successors to primitive subexpressions translated into jumps.

Synthesized attribute $jcond$ computes the correlation of the values of an expression with that of its rightmost identifier x.

Value of $jcond$ at expression e

$true$**:** The loaded value of x equals value of e,

$false$**:** The loaded value of x is negation of value of e.

Means for code generation:
Instruction following **LOAD** x is conditional jump to true–successor

JUMPT if $jcond = true$,

JUMPF if $jcond = false$.

attribute grammar BoolExp

nonterminals *IFSTAT, STATS, E, T, F;*

attributes inh *tsucc, fsucc* **with** *E,T,F* **domain string***;*
syn *jcond* **with** *E,T,F* **domain bool***;*
syn *code* **with** *IFSTAT, E,T,F* **domain string***;*

rules
IFSTAT $\rightarrow$ **if** *E* **then** *STATS* **else** *STATS* **fi**
$E.tsucc = t$
$E.fsucc = e$

$IFSTAT.code = E.code \mathbin{++} gencjump(\textbf{not}\ E.jcond, e) \mathbin{++}$
$t\!: \mathbin{++} STATS_1.code \mathbin{++} genujump(f) \mathbin{++} e\!: \mathbin{++} STATS_2.code \mathbin{++} f\!:$

$E \rightarrow T$

$E \rightarrow E\ \textbf{or}\ T$
$E_1.fsucc = t$
$E_0.jcond = T.jcond$
$E_0.code = E_1.code \mathbin{++} gencjump(E_1.jcond, E_0.tsucc) \mathbin{++} t\!: \mathbin{++} T.code$

$T \rightarrow F$

$T \rightarrow T\ \textbf{and}\ F$
$T_1.tsucc = f$
$T_0.jcond = F.jcond$
$T_0.code = T_1.code \mathbin{++} gencjump(\textbf{not}\ T_1.jcond, T_0.fsucc) \mathbin{++} f\!: \mathbin{++} F.code$

$F \rightarrow (E)$

$F \rightarrow \textbf{not}\ F$

$F_1.tsucc = F_0.fsucc$
$F_1.fsucc = F_0.tsucc$
$F_0.jcond = \textbf{not } F_1.jcond$

$F \rightarrow \textbf{id}$
$F.jcond = true$
$F.code = \textbf{LOAD id}.identifier$

Auxilliary functions:

$genujump(l) = \textbf{JUMP } l$
$gencjump(jc, l) = \textbf{if } jc = true$
$\textbf{then JUMPT } l$
$\textbf{else JUMPF } l$
$\textbf{fi}$