Attribute Grammars

– Wilhelm/Maurer: Compiler Design, Chapter 9 –

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

Attribute Grammars

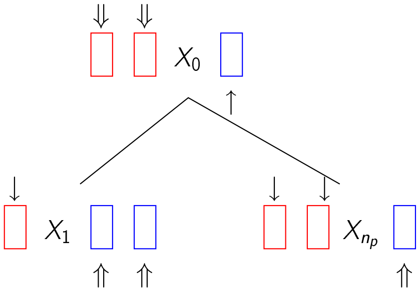
Attributes: containers for static semantic (non-context–free syntactic) information,

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

Describes nested scopes;

- a statement may be a block, consisting of a declaration aprt followed by a statement part,
- declaration parts consist of lists of procedure declarations,
- procedures, declared later in a list, may be called from within procedures declared earlier.

attribute grammar Scopes:
nonterminals Stms, Stm, Decls, Decl, Id, Args, Ptype;
domain Env = String → Types;
attributes **syn** ok **with** Decls, Decl, Stms, Stm **domain** Bool;
inh e-env **with** Stms, Stm, Decls, Decl **domain** Env;
inh it-env **with** Decls, Decl **domain** Env;
syn st-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.
A check for double declarations is made while collecting local declarations in it-env.

e-env is the "effective" environment, in which procedure calls are type checked.
For each nested scope, the effective environment is obtained by over-writing the external effective environment with the locally constructed environment.

rules

0 : $Stms \rightarrow Stm$

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

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

2 : $Stm \rightarrow$ **begin** $Decls\ ;\ Stms$ **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 : $Decls \rightarrow Decl$

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

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

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

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

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

5 : $Decl \rightarrow$ **proc** $Id : Ptype$ **is** $Stms$

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

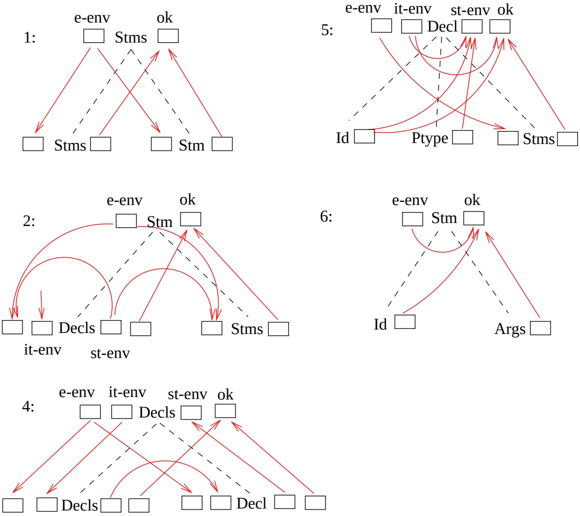
$Stms.e\text{-}env = Decl.e\text{-}env$

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

6 : $Stm \rightarrow$ **call** $Id\ (Args)$

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

Local Dependencies in the Scopes-AG



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.
  - $O(p)$ is the set of all attribute occurrences in production p.

Attribute Grammars – Terminology cont'd

- the association of a **domain** D_a with each attribute a;
- a **semantic rule**

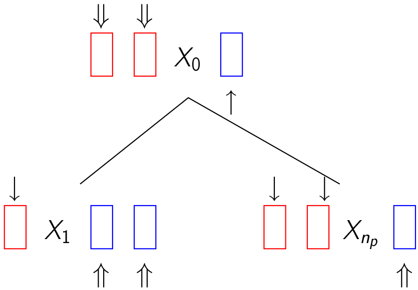
$$a_i = f_{p,a,i}\,(\,b^1_{j_1}, \ldots, b^k_{j_k}\,) \qquad (0 \le j_l \le n_p)\;(1 \le l \le k)$$

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

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

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

Attributes as Carriers of Context Information



Inherited

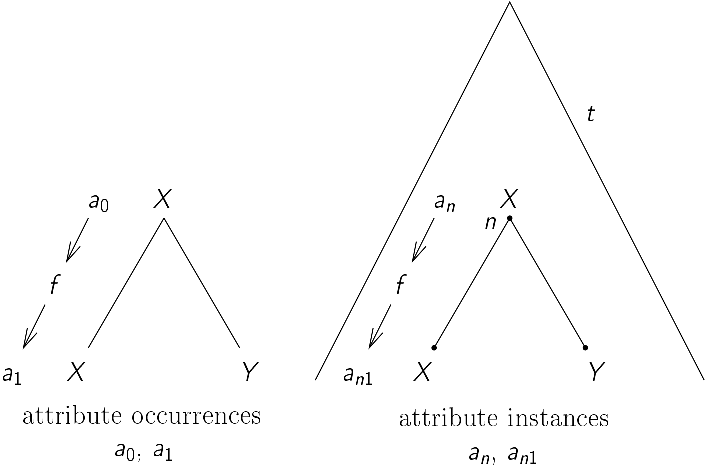
Synthesized

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

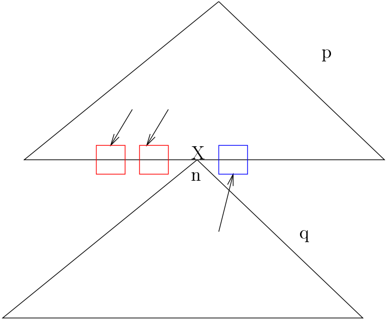


A production and one of its instances

The p–n–q Situation

Attribute evaluation at node n labelled X is determined by productions

- p applied at *parent(n)* for the inherited attributes of X and
- q applied at n for the synthesized attributes of X.



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 would in general 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 $1 \leq i \leq n_p$ or $a \in Syn(X_0)$ 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 → **if** E **then** STATS **else** STATS **fi**

E.tsucc = t

E.fsucc = e

IFSTAT.code = E.code ++ gencjump (**not** E.jcond, *e*) ++

t: ++ STATS_1.code ++ genujump (*f*) ++ e: ++ STATS_2.code ++ f:

E → T

E → E **or** T

E_1.fsucc = t

E_0.jcond = T.jcond

E_0.code = E_1.code ++ gencjump (E_1.jcond, E_0.tsucc) ++ t: ++ T.code

T → F

T → T **and** F

T_1.tsucc = f

T_0.jcond = F.jcond

T_0.code = T_1.code ++ gencjump (**not** T_1.jcond, T_0.fsucc) ++ f: ++ F.code

F → (*E*)

F → **not** F

F_1.tsucc = F_0.fsucc

F_1.fsucc = F_0.tsucc

F_0.jcond = **not** F_1.jcond

F → **id**

F.jcond = true

F.code = **LOAD id**.identifier

Auxilliary functions:

genujump (l) = **JUMP** l
gencjump (jc, l) = **if** jc = true
then JUMPT l
else JUMPF l
fi