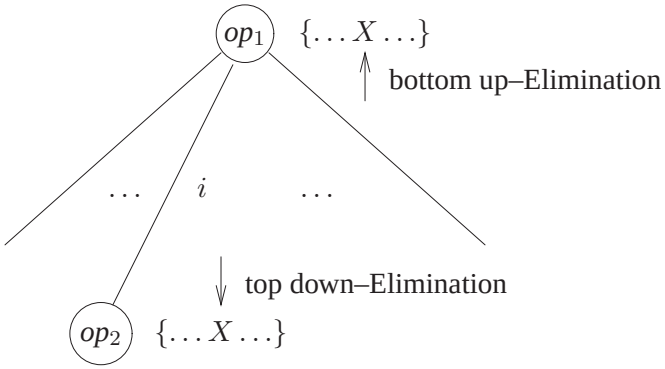
bottom_up_elim(*root*)*;*
top_down_elim(*root*)*;*
check whether now all *ops* sets have exactly one element; otherwise report an error
end

It looks like the bottom-up elimination and the top-down elimination do the same thing. This is almost correct. Figure 4.5 shows a combination of op_1- and op_2 labeled nodes. Each node is associated with the set of potential definitions of the operator. Bottom-up elimination possibly eliminates candidates from the set of definitions of op_1, top-down elimination from the set of definitions of op_2.

Fig. 4.5. The elimination of potential operators in two directions, bottom up and top down



4.2 Attribute Grammars

We have described tasks to be performed by semantic analysis in Section 4.1. In line with our presentation so far it would be nice to also have a description mechanism for these tasks, from which implementations could be generated.

Remember the algorithm for overload resolution in Section 4.1.3, in particular the two passes bottom-up elimination and top-down elimination. One step of bottom-up elimination at some node k removes from the set $ops(k)$ of potential operators at k all those where the type of the ith parameter does not agree with the result type of any potential operator at node $k.i$. Hence, the new set of potential operators at node k is determined based on the set of potential operators at the children of k. One step of top-down elimination at some node k removes operator candidates at $k.i$ whose result type does not agree with any of the parameter types of parameter i of the operator candidates at k. Here, the set of potential operators at $k.i$ is computed based on the set of potential operators at k. The overload-resolution algorithm needs this flow of information in both directions.

An elegant and powerful description mechanism for tasks like these are *attribute grammars*. They extend context-free grammars by associating *attributes* with the symbols of the underlying context-free grammar. These attributes are containers for static semantic information. Their values are computed by computations performed on trees, with computations traversing the trees as needed.

The set of attributes of a symbol X is denoted by $\mathcal{A}(X)$. With each attribute a is associated a type τ_a, which fixes the set of possible values for the instances of the attribute.

Consider a production $p: \ X_0 \longrightarrow X_1 \ldots X_k$ with $k \geq 0$ symbols occurring on the right side. To tell the different occurrences of symbols in production p apart, we number these from left to right. The left side nonterminal X_0 will be denoted by $p[0]$, the ith symbol X_i on the right side of p by $p[i]$ for

$i = 1, \ldots, k$. The attribute a of a symbol X has an *attribute occurrence* at each occurrence of X in a production. The occurrence of an attribute a at a symbol X_i is denoted by $p[i].a$.

Attributes also have *directions*, which can be better understood, if we think of production applications in parse trees. The attributes of a symbol X are either *inherited* or *synthesized*. The values of (instances of) synthesized attributes at a node are computed from (values of attribute instances in) the subtree at this node. The values of (instances of) inherited attributes at a node are computed from the context of the node, see Figure 4.6. All the attribute instances with ingoing yellow edges are computed within the production. These are the occurrences of synthesized attributes of the left side of the production and the occurrences of inherited attributes of the right side of productions. Together we call them *defining occurrences* of attributes in this production. All other occurrence of attributes in the production are called *applied attribute occurrences*. Each production has *semantic rules*, which describe how the values of defining attribute occurrences in the production are computed from the values of other attribute occurrences of the same production. So, semantic rules need to be given for each inherited attribute occurrence on the right side of a production and each synthesized attribute occurrence on the left side. The set of inherited and synthesized attributes of an attribute grammar are denoted by $\mathcal{I}$ and $\mathcal{S}$, resp. and the set of inherited and synthesized attributes of a symbols X correspondingly by $\mathcal{I}(X)$ and $\mathcal{S}(X)$.

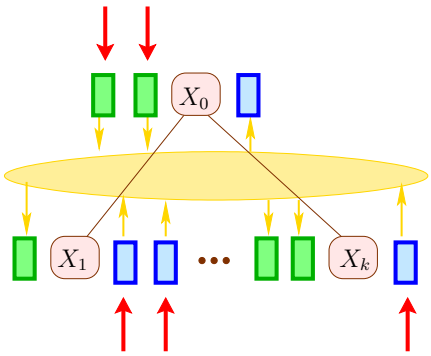


Fig. 4.6. An attributed node in the parse tree with its attributed successors. Instances of inherited attributes are drawn as boxes to the left of syntactic symbols, instances of synthesized attributes as boxes to the right of symbols. Red (darker) arrows show the information flow into the production instance from the outside, yellow (lighter) arrows symbolize functional dependences between attribute instances that are given through the semantic rules associated with the production.

In our examples we write the semantic rules in an OCAML-like programming language.

Popular LR parsers such as YACC and BISON offer a restricted attribute mechanism: Each symbol of the grammar has one associated attribute, and to each production there is one semantic rule that describes how the value of the attribute occurrence on the left side is computed from the values of the right-side attribute occurrences.

Example 4.2.1 Consider a context-free grammar with the nonterminals E, T, F for arithmetic expressions. The set of terminals consists of parentheses, operators, and the symbols var and const, representing *int* variables and constants. The nonterminals have an attribute *tree* that will hold the trees for the words that have been derived from them.

The productions of the grammar are extended by semantic rules as follows: Different occurrences of the same symbol are indexed.

Example 4.2.2

$$
\begin{aligned}
p_1: \quad & E \longrightarrow E + T \\
& \quad E[0].tree = \mathsf{Plus}\,(E[1].tree, T.tree) \\
p_2: \quad & E \longrightarrow T \\
& \quad E.tree = T.tree \\
p_3: \quad & T \longrightarrow T * F \\
& \quad T[0].tree = \mathsf{Mult}\,(T[1].tree, F.tree) \\
p_4: \quad & T \longrightarrow F \\
& \quad T.tree = F.tree \\
p_5: \quad & F \longrightarrow \mathsf{const} \\
& \quad F.tree = \mathsf{Int}\,(\mathsf{const}.val) \\
p_6: \quad & F \longrightarrow \mathsf{var} \\
& \quad F.tree = \mathsf{Var}\,(\mathsf{var}.id) \\
p_7: \quad & F \longrightarrow (\,E\,) \\
& \quad F.tree = E.tree
\end{aligned}
$$

□

The trees are built with the constructors Plus, Mult, Int, Var. Further, we assume that the symbol const has an attribute val, which will hold the value of the instances of const, and the symbol var has an attribute id, which will hold the unique codes for the instances of id. □

If the semantic rule for a defining attribute occurrence uses an attribute occurrence as argument there is a *functional dependence* from the argument attribute-occurrence to the defining attribute occurrence.

Attributes of symbols labeling nodes of parse trees are called *attribute instances*. They exist at compile time after syntactic analysis has produced the parse tree.

The functional dependences between attribute occurrences determine in which order the attribute instances at nodes of the parse tree may be evaluated. Arguments of semantic rules need to be evaluated before the rules can be applied to compute the value of the associated attribute (instance). There exist some constraints on the functional dependences to ensure that the *local* semantic rules of the attribute grammar for the attribute occurrences in the productions can be composed to a *global* computation of all attribute instances in a parse tree. The values of attribute instances at the individual nodes of the parse tree are computed by a global algorithm, which is generated from the attribute grammar, and which at each node n adheres to the local semantic rules of the production applied at n. The theme of this chapter is how such an algorithm can be automatically generated from a given attribute grammar.

An attribute grammar is in *normal form* if all defining attribute occurrences in productions only depend on applied occurrences in the same productions. If not explicitly stated otherwise we assume that attribute grammars are in normal form.

We have admitted synthesized attributes for terminal symbols of the grammar. In compiler design, attribute grammars are used to specify semantic analysis. This phase follows lexical and syntactic analysis. Typical synthesized attributes of terminal symbols are values of constants, external representations or unique encodings of names, and the addresses of string constants. The values of these attributes are delivered by the scanner, at least if it is extended by semantic functionality. So, in compilers synthesized attributes of terminal symbols play an important role. Also inherited attribute instances at the root of the parse tree have no semantic rules in the grammar to compute their values. However, the compiler may have some use for them and therefore will initialize them.

4.2.1 The Semantics of an Attribute Grammar

The semantics of an attribute grammar determines for each parse tree t of the underlying context-free grammar which values the attributes at each node in t should have.

For each node n in t let $\mathsf{symb}(n)$ the symbol of the grammar labeling n. If $\mathsf{symb}(n) = X$ then n is associated with the attributes in $\mathcal{A}(X)$. The attribute $a \in \mathcal{A}(n)$ of the node n addressed by $n.a$. Furthermore we need an operator to navigate from a node to its successors. Let $n_1, \ldots, n_k$ be the

sequence of successors of node n in parse tree t. $n[0]$ denotes the node n itself, and $n[i] = n_i$ for $i = 1, \ldots, k$ denotes the $i$thn successor of n in the parse tree t.

If $X_0 = \mathsf{symb}(n)$ and, if $X_i = \mathsf{symb}(n_i)$ for $i = 1, \ldots, k$ are the labels of the successors n_i of n then $X_0 \longrightarrow X_1 \ldots X_k$ is the production of the context-free grammar that was applied at node n. The semantic rules of this production p provide the method to compute values of the attributes at the nodes $n, n_1, \ldots, n_k$. The semantic rule

$$p[i].a = f(p[i_1].a_1, \ldots, p[i_r].a_r)$$

for the production p becomes the semantic rule

$$n[i].a = f(n[i_1].a_1, \ldots, n[i_r].a_r)$$

for the node n in the parse tree. We assume that the semantic rules specify *total* functions. Let t be a parse tree and

$$V(t) = \{n.a \mid n \text{ node in } t, a \in \mathcal{A}(\mathsf{symb}(n))\}$$

be the set of all attribute instances in t. The subset $V_{in}(t)$ of inherited attribute instances at the rootl and of synthesized attribute instances at the leaves are called the set of *input attribute instances* of t.

Assigning values to the input attribute instances and instantiating the semantic rules of the attribute grammar at all nodes in t produces a system of equations in the unknowns $n.a$ that has for all but the input attribute instances exactly one equation. Let $\mathsf{AES}(t)$ this attribute equation system. If $\mathsf{AES}(t)$ is recursive (cyclic) it can have several solutions or no solution. IF $\mathsf{AES}(t)$ is not recursive there exists for assignment σ of the input attribute instance exactly one attribute assignment for the parse trees t agreeing on the input attribute instances with σ and satisfying all equations. The attribute grammar is therefore called *wellformed*, if the system of equations $\mathsf{AES}(t)$ is not recursive for any parse tree t of the underlying context-free grammar. In this case we define the semantics of the attribute grammar as the function mapping each parse tree t and each assignment σ of the input attribute instances to an attribute assignment that agrees on these with σ and that additionally satisfies all equations of the system $\mathsf{AES}(t)$.

4.2.2 Some Attribute Grammars

In the following we present some attribute grammars that solve essential subtasks of semantic analysis. The first attribute grammar shows how types of expressions can be computed using an attribute grammar.

Example 4.2.3 (Type checking) The attribute grammar AG_{types} beschreibt die type determination for expressions containing assignments, nullary functions, operators $+, -, *, /$ and variable and constants of type **int** or **float**in a C-like programming language with explicit type declarations for variables. The attribute grammar has an attribute typ for the nonterminal symbols E, T and F and for the terminal symbol const, which may take values Int and Float. This grammar can be easily extended to more general expressions with function application, component selection in composed values, and pointers.

$$
\begin{array}{l}
E \longrightarrow \mathsf{var}\ '='\ E \\
E[1].env = E[0].env \\
E[0].typ\ = E[0].env\ \mathsf{var}.id \\
E[0].ok\ \ = \quad \mathbf{let}\ x = \mathsf{var}.id \\
\qquad\quad \mathbf{in\ let}\ \tau = E[0].env\, x \\
\qquad\quad \mathbf{in}\quad (\tau \neq \mathsf{error}) \wedge (E[1].type \sqsubseteq \tau)
\end{array}
$$

$$
\begin{array}{l}
E \longrightarrow E\ \mathsf{aop}\ T \\
E[1].env = E[0].env \\
T.env\ \ \ = E[0].env \\
E[0].typ\ = E[1].typ \sqcup T.typ \\
E[0].ok\ \ = (E[1].typ \sqsubseteq \mathsf{float}) \wedge (T.typ \sqsubseteq \mathsf{float})
\end{array}
\qquad
\begin{array}{l}
E \longrightarrow T \\
T.env = E.env \\
E.typ = T.typ \\
E.ok\ \ = T.ok
\end{array}
$$

$$
\begin{array}{l}
T \longrightarrow T\ \mathsf{mop}\ F \\
T[1].env = T[0].env \\
F.env\ \ \ = T[0].env \\
T[0].typ\ = T[1].typ \sqcup F.typ \\
T[0].ok\ \ = (T[1].typ \sqsubseteq \mathsf{float}) \wedge (F.typ \sqsubseteq \mathsf{float})
\end{array}
\qquad
\begin{array}{l}
T \longrightarrow F \\
F.env = T.env \\
T.typ\ = F.typ \\
T.ok\ \ = F.ok
\end{array}
$$

$$
\begin{array}{l}
F \longrightarrow (E) \\
E.env = F.env \\
F.typ\ = E.typ \\
F.ok\ \ = E.ok
\end{array}
\qquad
\begin{array}{l}
F \longrightarrow \mathsf{const} \\
F.typ = \mathsf{const}.typ \\
F.ok\ \ = \mathsf{true}
\end{array}
$$

$$
\begin{array}{l}
F \longrightarrow \mathsf{var} \\
F.typ = F.env\ \mathsf{var}.id \\
F.ok\ \ = (F.env\ \mathsf{var}.id \neq \mathsf{error})
\end{array}
\qquad
\begin{array}{l}
F \longrightarrow \mathsf{var}\,() \\
F.typ = (F.env\ \mathsf{var}.id)\,() \\
F.ok\ \ = \mathbf{match}\ F.env\ \mathsf{var}.id \\
\qquad\quad \mathbf{with}\ \tau\,() \rightarrow \mathsf{true} \\
\qquad\quad\ \ |\quad \_\quad \rightarrow \mathsf{false}
\end{array}
$$

The attribute env of the nonterminals E, T and F is inherited, all other attributes of grammar AG_{types} are synthesized.

The semantic rules in this grammar are not necessarily *total* functions. For instance, they are not total if an identifier is used, but not declared in a program. In this case, an entry for this identifier would be searched for in the symbol table env, Also, the right side of the semantic rule to the third production produces the value Int for the operator $'\div'$ only if the attribute occurrence $T[1].typ$ and $F.typ$ both have the value Int. Otherwise it is not defined.

If a semantic rule is undefined for particular arguments this is understood as returning an error value. No operator is defined for such an error value. Thus, the error value propagates. The compiler would reject a program with error values in attributes. □

We use a convention for writing attribute grammars to reduce the writing effort caused mainly by chain productions:

If no semantic rule for a defining occurrence is given the identity function as semantic rule is assumed.

The following examples use this convention, at least for rules $A \longrightarrow \alpha X \beta$ whose right sides contain only one symbol X whose attributes agree with the ones of the left side A.

Example 4.2.4 (Managing Symbol Tables) Attribute grammar AG_{scopes} manages symbol tables for a fragment of a C-like imperative language with parameterless procedures. Nonterminals for declarations, statements, blocks, and expressions are associated with an with a inherited attribute env that will contain the actual symbol table.

The redeclaration of an identifier within the same block is forbidden while it is allowed in a new block. To check this a further inherited attribute $same$ is used to collect the set of identifiers that are encountered so far in the actual block. The synthesized attribute ok signals whether all used identifiers are declared and used in a type-correct way.

$\langle decl \rangle \longrightarrow \langle type \rangle\ \mathsf{var};$

$\langle decl \rangle.new = (\mathsf{var}.id, \langle type \rangle.typ)$
$\langle decl \rangle.ok = \mathsf{true}$

$\langle decl \rangle \longrightarrow \mathsf{void\ var\ (\ )\ \{}\ \langle block \rangle\ \mathsf{\}}$

$\langle block \rangle.same = \emptyset$
$\langle block \rangle.env = \langle decl \rangle.env \oplus \{\mathsf{var}.id \mapsto \mathsf{void}\,()\}$
$\langle decl \rangle.new = (\mathsf{var}.id, \mathsf{void}\,())$
$\langle decl \rangle.ok = \langle block \rangle.ok$

$\langle block \rangle \longrightarrow \langle decl \rangle\ \langle block \rangle$

$\langle decl \rangle.env = \langle block \rangle[0].env$
$\langle block \rangle[1].same = \mathbf{let}\ (x, \_) = \langle decl \rangle.new\ \mathbf{in}\ \langle block \rangle[0].same \cup \{x\}$
$\langle block \rangle[1].env = \mathbf{let}\ (x, \tau) = \langle decl \rangle.new\ \mathbf{in}\ \langle block \rangle[0].env \oplus \{x \mapsto \tau\}$
$\langle block \rangle[0].ok = \mathbf{let}\ (x, \_) = \langle decl \rangle.new\ \mathbf{in}\ \mathbf{if}\ \neg(x \in \langle block \rangle[0].same)\ \mathbf{then}\ \langle decl \rangle.ok \wedge \langle block \rangle[1].ok\ \mathbf{else}\ \mathsf{false}$

$\langle stat \rangle \longrightarrow E;$

$E.env = \langle stat \rangle.env$
$\langle stat \rangle.ok = E.ok$

$\langle stat \rangle \longrightarrow \mathsf{\{}\ \langle block \rangle\ \mathsf{\}}$

$\langle block \rangle.env = \langle stat \rangle.env$
$\langle block \rangle.same = \emptyset$
$\langle stat \rangle.ok = \langle block \rangle.ok$

$\langle block \rangle \longrightarrow \langle stat \rangle\ \langle block \rangle$

$\langle stat \rangle.env = \langle block \rangle[0].env$
$\langle block \rangle[1].env = \langle block \rangle[0].env$
$\langle block \rangle[1].same = \langle block \rangle[0].same$
$\langle block \rangle[0].ok = (\langle stat \rangle.ok \wedge \langle block \rangle[1].ok)$

$\langle block \rangle \longrightarrow \epsilon$

$\langle block \rangle.ok = \mathsf{true}$

This grammar only contains productions for the nonterminal symbol $\langle stat \rangle$. To obtain a complete grammar further productions for expressions like the ones in 4.2.3 are needed. For the case that the programming language also contains type declarations another attribute is required that manages the actual type environment.

The given rules collect declarations from left to right. This excludes the use of a procedure *vor* before its declaration. This formalizes the scoping rule for the language, namely that the scope of a procedure declaration begins at the end of the declaration. We want now to change this scoping rules to allow the use of procedures starting with the beginning of the block in which they are declared The modified attribute grammar reflecting this is called $AG_{scopes+}$. In the attribute grammar $AG_{scopes+}$ the computation of the attribute env is modified such that all procedures declared in the block are added to env already at the beginning of a block. The nonterminal $\langle block \rangle$ is associated with an additional synthesized attribute $procs$, and the productions for the nonterminal $\langle block \rangle$ obtain the additional rules:

$$\langle block\rangle \longrightarrow \varepsilon$$
$$\langle block\rangle.procs = \emptyset$$

$$\langle block\rangle \longrightarrow \langle stat\rangle\ \langle block\rangle$$
$$\langle block\rangle[0].procs = \langle block\rangle[1].procs$$

$$\langle block\rangle \longrightarrow \langle decl\rangle\ \langle block\rangle$$
$$\begin{aligned}\langle block\rangle[0].procs = \ & \mathbf{match}\ \langle decl\rangle.new \\ & \mathbf{with}\ (x, \mathsf{void}()) \rightarrow \langle block\rangle[1].procs \oplus \{x \mapsto \mathsf{void}()\} \\ & \quad \mid \_ \qquad\qquad\ \rightarrow \langle block\rangle[1].procs\end{aligned}$$

The procedures collected in $\langle block\rangle.procs$ are added to the environment $\langle block\rangle.env$ in the productions that introduce new blocks. The attribute grammar $AG_{scopes+}$ then has the following semantic rules:

$$\langle stat\rangle \longrightarrow \{\ \langle block\rangle\ \}$$
$$\langle block\rangle.env = \langle stat\rangle.env \oplus \langle block\rangle.procs$$

$$\langle decl\rangle \longrightarrow \mathsf{void\ var}\ ()\ \{\ \langle block\rangle\ \}$$
$$\langle block\rangle.env = \langle decl\rangle.env \oplus \langle block\rangle.procs$$

The rest of attribute grammar $AG_{scopes+}$ agrees with attribute grammar AG_{scopes}. Note that the new semantic rules induce an interesting functional dependency: Inherited attributes of a nonterminal on the right side of a production depend on synthesized attributes of the same nonterminal. □

Attribute grammars can be used to generate code. The code-generation functions as they were described in *Wilhelm/Seidl: Compiler Design—Virtual Machines* are recursively defined over the structure of programs. They use information about the program such as the types of identifiers visible in a program fragment whose computation can be described by attribute grammars. We now give an example how a nontrivial subproblem of code generation can be described by an attribute grammar. This is the so-called *short-circuit evaluation* of boolean expressions.

Example 4.2.5 We consider code generation for a virtual machine like the CMA in *Wilhelm/Seidl: Compiler Design—Virtual Machines*,

The code generated for a boolean expression according to attribute grammar *BoolExp* should have the following properties:

- the generated code consists only of load-instructions and conditional jumps. In particular, no boolean operations are generated.
- Subexpressions are evaluated from left to right.
- Of each subexpression as well of the whole expression only the smallest subexpressions are evaluated that uniquely determine the value of the whole (sub)expression. So, each subexpression is left as soon as its value determines the value of its containing expression.

The following code is generated for the boolean expression $(a \wedge b) \vee \neg c$ with the boolean variables a, b and c:

```
        load a
        jumpf l1                          jump-on-false
        load b
        jumpt l2                          jump-on-true
l1:     load c
        jumpt l3
l2:     continuation if the expression evaluates to true
l3:     continuation if the expression evaluates to false
```

The attribute grammar *BoolExp* generates labels for the code for subexpressions, and it transports these labels to atomic subexpressions from which the evaluation jumps to these labels. Each subexpression E and T receives in $fsucc$ the label of the successor if the expression evaluates to false, and in $tsucc$ the label of the successor if it evaluates to true. A synthesized attribute $jcond$ contains the relation of the value of the whole (sub)expression to its rightmost identifier.

- If $jcond$ has the value true for an expression this means that the value of the expression is the same as the value of its rightmost identifier. This identifier is the last one that is loaded during the evaluation.
- If $jcond$ has the value false the value of expression is the negation of the value of its rightmost identifier.

Correspondingly, a *load* instruction for the last identifier is followed by a jumpt to the label in $tsucc$, if $jcond = \mathsf{true}$, and it is followed by a jumpf if $jcond = \mathsf{false}$. This selection is performed by the function:

$$\mathsf{gencjump}\,(jc, l) = \mathbf{if}\ jc\ \mathbf{then}\ (\mathsf{jumpt}\,l)\ \mathbf{else}\ (\mathsf{jumpf}\,l)$$

As a context for boolean expressions we add a production for a two-sided conditional statement. The labels $tsucc$ and $fsucc$ of the condition quite naturally correspond to the start addresses of the code for the *then* and the *else* parts of the conditional statements. The code for the condition ends in a conditional jump to the *else* part. It tests the condition E for the value false. Therefore the function gencjump receives $\neg jcond$ as first parameter. We obtain:

$$
\begin{array}{lll}
\langle if\_stat\rangle \longrightarrow \mathsf{if}\ (E)\ \langle stat\rangle\ \mathsf{else}\ \langle stat\rangle & & \\
\quad E.tsucc & = \mathsf{new}() & \\
\quad E.fsucc & = \mathsf{new}() & \\
\quad \langle if\_stat\rangle.code = & \mathbf{let}\ t = E.tsucc & \\
 & \mathbf{in\ let}\ e = E.fsucc & \\
 & \mathbf{in\ let}\ f = \mathsf{new}() & \\
 & \mathbf{in}\ E.code\ \hat{}\ \mathsf{gencjump}\,(\neg E.jcond, e)\hat{} & \\
 & \quad t : \hat{}\ \langle stat\rangle[1].code\hat{}\ \mathsf{jump}\ f\hat{} & \\
 & \quad e : \hat{}\ \langle stat\rangle[2].code\hat{} & \\
 & \quad f : & \\
E \longrightarrow T & & \\
E \longrightarrow E\ \mathsf{or}\ T & & \\
\quad E[1].tsucc & = E[0].tsucc & T.tsucc = E[0].tsucc \\
\quad E[1].fsucc & = \mathsf{new}() & T.fsucc = E[0].fsucc \\
\quad E[0].jcond & = T.jcond & \\
\quad E[0].code & = \mathbf{let}\ t = E[1].fsucc & \\
 & \mathbf{in}\ E[1].code\hat{}\ \mathsf{gencjump}\,(E[1].jcond, E[0].tsucc)\hat{} & \\
 & \quad t : \hat{}\ T.code & \\
T \longrightarrow F & & \\
T \longrightarrow T\ \mathsf{and}\ F & & \\
\quad T[1].tsucc & = \mathsf{new}() & F.tsucc = T[0].tsucc \\
\quad T[1].fsucc & = T[0].fsucc & F.fsucc = T[0].fsucc \\
\quad T[0].jcond & = F.j\,cond & \\
\quad T[0].code & = \mathbf{let}\ f = T[1].tsucc & \\
 & \mathbf{in}\ T[1].code\hat{}\ \mathsf{gencjump}\,(\neg T[1].jcond, T[0].fsucc)\hat{} & \\
 & \quad f : \hat{}\ F.code & \\
F \longrightarrow (E) & & \\
F \longrightarrow \mathsf{not}\ F & & \\
\quad F[1].tsucc & = F[0].fsucc & \\
\quad F[1].fsucc & = F[0].tsucc & \\
\quad F[0].code & = F[1].code & \\
\quad F[0].jcond & = \neg F[1].jcond & \\
F \longrightarrow \mathsf{var} & & \\
\quad F.jcond & = \mathsf{true} & \\
\quad F.code & = \mathsf{load\ var}.id &
\end{array}
$$

The infix operator $\hat{}$ denotes the concatenation of code fragments. This attribute grammar is not in normal form: The semantic rule for the synthesized attribute $code$ of the left side $ifstat$ in the first production uses the inherited attributes $tsucc$ and $fsucc$ of the nonterminal E on the right side. The reason is that the two inherited attributes are computed using a function $\mathsf{new}()$ that generated a new label every time it is called. This way it changes a global state, which is, puristically seen, not admitted by the attribute-grammar formalism. □

4.3 The Generation of Attribute Evaluators

This section treats attribute evaluation, more precisely the evaluation of attribute instances in parse trees, even more precisely the generation of the corresponding evaluators.

An attribute grammar defines for each parse tree t of the underlying context-free grammar a system of equations $\mathsf{AES}(t)$. The unknowns in this system of equations are the attribute instances at the nodes of t. Let us assume that the attribute grammar is well-formed. In this case the system of equations is not recursive. Non-recursive systems of equations can be solved by elimination methods. Each elimination step selects one attribute instance to be evaluated next. It must only depend on already evaluated attribute instances. Such an attribute evaluator is purely *dynamic* as it does not exploit any information about the dependences in the attribute gammar. An evaluator that makes use of such information is described in the next section.

4.3.1 Demand-driven Attribute Evaluation

We now describe a first, dynamic attribute evaluator for well-formed attribute grammars, which evaluates attribute instances in a *demand-driven* way.

Demand-driven evaluation means that not all attribute instances will get their values, but attribute evaluation will be triggered by a *value enquiry* for some attribute instances. This demand-driven evaluation is performed by a function solve, which is called for a node and one of the attributes a of the symbol that labels n. The evaluation starts by checking whether the demanded attribute instance $n.a$ has already received its value. If this is the case the function returns with the already computed value. Otherwise the evaluation of $n.a$ is triggered. This evaluation may in turn demand the evaluation of other attribute instances, whose evaluation is triggered recursively. This strategy has the consequence that for each attribute instance in the parse tree the right side of its semantic rule is evaluated at most once. The evaluation of attribute instances that are never demanded is completely avoided.

To realize this idea all attribute instances that are not initialized are set to the value Undef before the first value enquiry. Each attribute instance initialized with a non-Undef value d is set to the value Value d. For the navigation in the parse tree we use the postfix operators $[i]$ to go from a node n to its ith successor. For $i = 0$ the navigation stays at n. Furthermore we need an operator father when given a node n returns the (n', j) consisting of the father n' of node n and the information in which direction, seen from n', to find n. This latter information says which child the argument node is of its father n'. To implement the function solve for this recursive evaluation, we need a function eval. If p is the production that was applied at node n, and if

$$f(p[i_1].a_1, \ldots, p[i_r].a_r)$$

is the right side of the semantic rule for the attribute occurrence $p[i].a$, eval $n\ (i, a)$ returns the value of f, where for each demanded attribute instance the function solve is called, that is,

$$\mathsf{eval}\ n\ (i, a) = f(\mathsf{solve}\ n[i_1]\ a_1, \ldots, \mathsf{solve}\ n[i_r]\ a_r)$$

In a simultaneous recursion with the function eval the function solve is implemented by:

$$\begin{array}{l}
\mathsf{solve}\ n\ a = \mathbf{match}\ n.a \\
\quad \mathbf{with}\ \mathsf{Value}\ d \to d \\
\quad\ \mid\ \ \mathsf{Undef}\ \ \to \mathbf{if}\ b \in \mathcal{S}(\mathsf{symb}(n)) \\
\qquad\qquad\qquad\qquad \mathbf{then}\ \ \mathbf{let}\ d = \mathsf{eval}\ n\ (0, a) \\
\qquad\qquad\qquad\qquad\qquad\ \mathbf{in\ let}\ _ = n.a \leftarrow \mathsf{Value}\ d \\
\qquad\qquad\qquad\qquad\qquad\ \mathbf{in}\ d \\
\qquad\qquad\qquad\qquad \mathbf{else}\ \ \ \mathbf{let}\ (n', j') = \mathsf{father}\ n \\
\qquad\qquad\qquad\qquad\qquad\ \mathbf{in\ let}\ d' = \mathsf{eval}\ n'\ (j', a) \\
\qquad\qquad\qquad\qquad\qquad\ \mathbf{in\ let}\ _ = n.a \leftarrow \mathsf{Value}\ d' \\
\qquad\qquad\qquad\qquad\qquad\ \mathbf{in}\ d'
\end{array}$$

The function solve checks, whether the attribute instance $n.a$ in the parse tree already has a value. If this is the case solve returns this value. If the attribute instance $n.a$ does not yet have a value, $n.a$ is labeled with Undef. In this case the semantic rule for $n.a$ is searched for.

If a is a synthesized attribute of the symbol at node n, there is a semantic rule to the production p at node n. The right side f of this rule is modified such that it does not directly attempt to access its argument attribute instances, but instead calls the function solve recursively for these instances for node n. If a value d for the attribute instance $n.a$ is obtained, it is assigned to the attribute instance $n.a$ and in addition returned as result.

If a is an inherited attribute of the symbol at node n, the semantic rule for $n.a$ is not supplied by n, but by the father of n. Let n' be the father of n and n the j'th child of n'. The semantic rule for the attribute occurrence $p'[j'].a$ is chosen if the production p' was applied at node n'. Its right side is again modified in the same way such that before any access to attribute values the function solve is called for the node n'. The computed value is stored in the attribute instance $n.a$ and returned as result.

If the attribute grammar is well-formed, the demand-driven evaluator computes for each parse tree and for each attribute instance always the correct value. If the attribute grammar is not well-formed there exist parse trees for which the associated system of equations is recursive. If t is such a parse tree, there is in t a node n and an attribute a at n such that $n.a$ depends, directly are indirectly, on itself. The call solve n a might then possibly not terminate. This nontermination can be avoided by labeling attribute instances with Called if their evaluation has started, but not yet terminated. The function solve would terminate evaluation when it meets such an attribute instance labeled with Called. solve would return an error value in this case, see Exercise **??**.

4.3.2 Static Precomputations for Attribute Evaluators

Dynamic attribute evaluation do not exploit information about the attribute grammar to improve the efficiency of attribute evaluation. More efficient attribute evaluation methods is possible using knowledge of the functional dependences in productions. An attribute occurrence $p[i].a$ in production p functionally depends on an occurrence $p[j].b$ if $p[j].b$ is an argument for the semantic rule for $p[i].a$.

These production-local dependences determine the dependences in the system of equation $\mathsf{AES}(t)$. The evaluation method that we describe now analyzes the functional dependences between attribute occurrences in productions to derive information about global dependences and a visit sequence for the attribute occurrence of each production that guarantees that an attribute instance is only visited when the argument instances for the corresponding semantic rule are already evaluated. Consider again Figure 4.6. Attribute evaluation requires a cooperation of the computations at a node n and and its successors $n_1, \ldots, n_k$, and those in the context of this production instance. A local computation of an instance of a synthesized attribute at a node n labeled with X_0 provides an attribute value to be used by local computations in the upper context of n. The computation of the value of an inherited attribute instance at the same node n takes place above n and provides a new attribute value available for local computations below n according to the semantic rules for the production $p: \; X_0 \longrightarrow X_1 \ldots X_k$. To schedule this interaction of computations *global* functional dependences between attribute instances need to be derive from production-local dependences. We introduce some notions:

For a production p let $V(p)$ be the set of attribute occurrences in p. The semantic rules to production p define a relation $Dp(p) \subseteq V(p) \times V(p)$ of *production-local* functional dependences on the set $V(p)$. The relation $Dp(p)$ contains a pair $(p[j].b, p[i].a)$ of attribute occurrences if and only if $p[j].b$ occurs as an argument in a semantic rule for $p[i].a$.

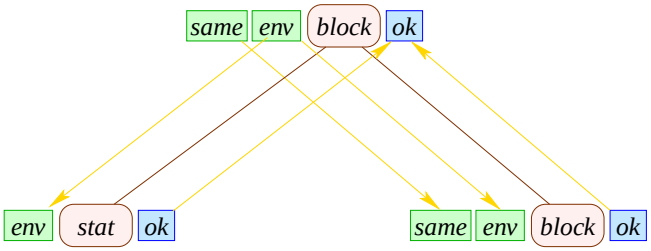


Fig. 4.7. The production-local dependence relation to production $block \rightarrow stat \;\; block$ in AG_{scopes}

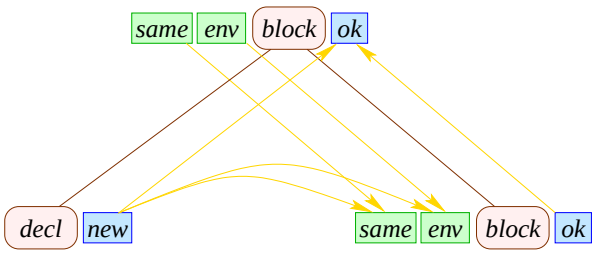


Fig. 4.8. The production-local dependence relation to production $block \longrightarrow decl\ block$ in AG_{scopes}.

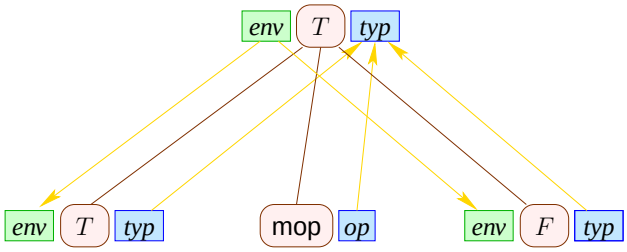


Fig. 4.9. Die production-local dependence relation zur production $T \longrightarrow T\ \mathsf{mop}\ F$ aus AG_{types}

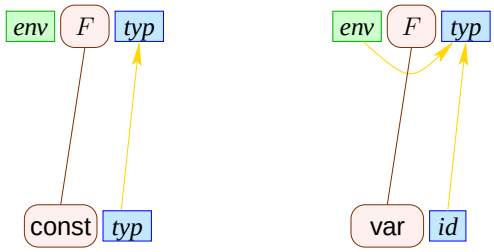


Fig. 4.10. The production-local dependence relations of the productions $F \longrightarrow \mathsf{const}$ and $F \longrightarrow \mathsf{var}$ in AG_{types}.

Example 4.3.1 (Continuation of Examples 4.2.4 and 4.2.3) To increase readability we represent attribute-dependence relations always together with the underlying syntactic structure, that is, the production or the parse tree. The dependence relations for the productions $block \longrightarrow stat\ \ block$ and $block \longrightarrow decl\ \ block$ in AG_{scopes} are shown in Figure 4.7 and 4.8. The production-local dependence relations for attribute grammar AG_{types} are all very simple: There are dependences between the occurrence of the inherited attribute env on the left side and the inherited occurrence of attribute env on the right side and between the synthesized attributes typ and op on the right side to the synthesized attribute typ on the left side (see Figure 4.9). Only in production $F \longrightarrow \mathsf{var}$ there is a dependence between the attributes env and typ of nonterminal F (see Figure 4.10). □

In attribute grammars in normal form the arguments of semantic rules for defining occurrences are always applied attribute occurrences. Therefore the paths in all production-local dependence relations have length 1, and there exist no cycles of the form $(p[i].a, p[i].a)$. The adherence to normal form simplifies some considerations. If not explicitly said otherwise we assume in the following that all attribute grammars are in normal form.

The production-local dependences between attribute occurrence in productions induce dependences between attribute instances in the parse trees of the grammar. Let t be a tree of the context-free grammar underlying an attribute grammar. The *individual* dependence relation on the set $V(t)$ of attribute instances of t, $Dt(t)$, is obtained by *instantiating* the production-local dependence relations of productions applied in t. For each node n in t at which production p has been applied the relation $Dt(t)$ consists of exactly the pairs $(n[j].b, n[i].a)$ with $(p[j].b, p[i].a) \in Dp(p)$.

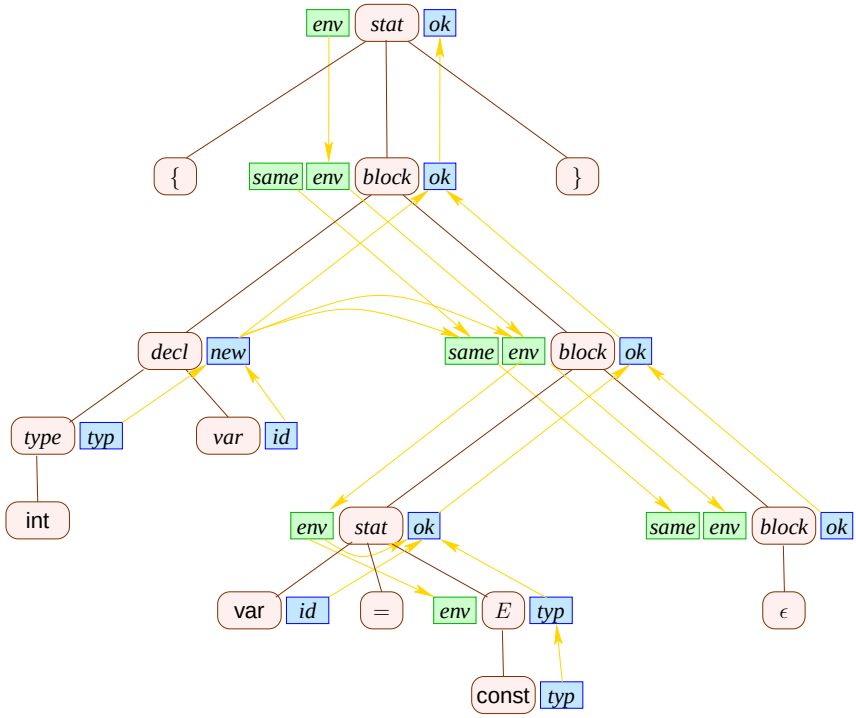


Fig. 4.11. The individual dependence relation for the parse tree to { int x; $x = 1$; } according to attribute grammar AG_{scopes}

Example 4.3.2 (Continuation of Example 4.2.4) The dependence relation to the parse tree of statement { **int** x; $x = 1$; } according to attribute grammar AG_{scopes} is shown in Figure 4.11. For simplicity we assumed that the nonterminal *type* directly derives the base type int, and that nonterminal E for expressions directly derives the terminal const. □

A relation R on a set A is called *cyclic* if its transitive closure contains (a, a). Otherwise we call the relation R *acyclic*. A attribute grammar is called *non-circular*, if all individual dependence relations of the attribute grammar are acyclic. An individual dependence relation $Dt(t)$ is acyclic if and only if the system of equations $\mathsf{AES}(t)$ that was introduced in Section 4.2.1 is not recursive. Attribute grammars that satisfied the latter condition were called well-formed. Thus, an attribute grammar is well-formed if and only it is well-formed.

Consider a parse tree t with root label X as in Figure 4.12. The instances of the inherited attributes at the root are viewed as input to t, and the instances of the synthesized attributes at the root as output of t. The instance of d at the root (transitively) depends only on the instance of c at the root. If the value of the instance of c is known an attribute evaluator can descend into t and return with the value for the instance of d since there are no other dependences of instances external to t that do not pass through c. The instance of e at the root depends on the instances of a and b at the root. When both values are available the evaluation of the instance of e can be triggered. This situation is described by the *lower characteristic dependence relation* of X induced by t. This is a relation over the set $\mathcal{A}(X)$.

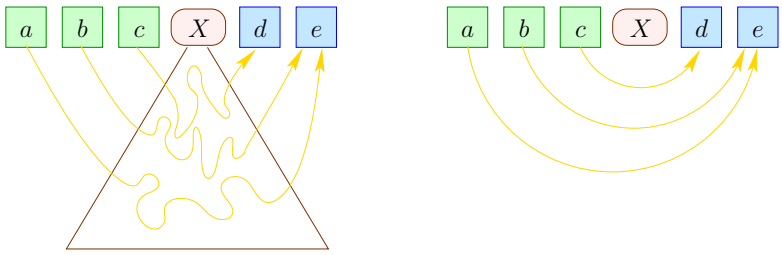


Fig. 4.12. Attribute dependences in a parse tree for X and the induced lower characteristic dependence relation

Let t be a parse tree for a symbol X with root n. The lower characteristic dependence relation $R_t(X)$ for X induced by t consists of all pairs (a, b) of attributes for which the pair $(n.a, n.b)$ of attribute instances at the root n of t is in the transitive closure of the individual dependence relation $Dt(t)$. In particular is

$$R_t(X) \subseteq \mathcal{I}(X) \times \mathcal{S}(X).$$

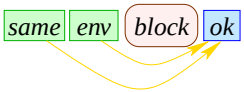


Fig. 4.13. Lower characteristic dependence relation for *block*

Example 4.3.3 (Continuation of Example 4.3.2) The lower characteristic dependence relation for the nonterminal *block* induced by the subtree of the root of t with root *block* in Example 4.3.2 is shown in Figure 4.13. □

Lemma 4.1. For an attribute grammar the following statements are equivalent:

1. For each parse tree t with root label X, the lower characteristic dependence relation $U_t(X)$ is acyclic;
2. For each parse tree t the dependence relation $Dt(t)$ is acyclic.

While the set of dependence relations of an attribute grammar is in general infinite the set of *lower* dependence relations is always finite since there is one such relation per nonterminal. One can thus compute the set of all lower dependence relations and then decide whether the dependence relations $Dt(t)$ for each parse tree t is acyclic, and whether the demand-driven attribute evaluator always terminates.

Let X be a symbol with a set $\mathcal{A}$ of attributes. For a relation $R \subseteq \mathcal{A}^2$ and $i \geq 0$ and a production p we define the relation $R[p, i]$ as

$$R[p, i] = \{(p[i].a, p[i].b) \mid (a, b) \in R\}$$

Consider a production $p : X \longrightarrow X_1 \ldots X_k$. For a binary relation $S \subseteq V(p)^2$ over the set of attribute occurrence of production p we define the following two operations

$$\begin{aligned} S^+ &= \bigcup\{S^j \mid j \geq 1\} && \text{(transitive closure)} \\ \pi_i(S) &= \{(a, b) \mid (p[i].a, p[i].b) \in S\} && \text{(projection)} \end{aligned}$$

Projection allows extracting the induced dependences between the attributes of a symbol occurring in a production p out of a dependence relation for attribute occurrences in p. We can thereby define the effect

$[\![p]\!]^\sharp$ of the application of production p on the dependence relations $R_1, \ldots, R_k$ for symbol occurrences $p[i]$ on the right side of p by:

$$[\![p]\!]^\sharp(R_1, \ldots, R_k) = \pi_0((Dp(p) \cup R_1[p,1] \cup \ldots \cup R_k[p,k])^+)$$

The operation $[\![p]\!]^\sharp$ takes the local dependence relation of production p and adds the instantiated dependence relations for the symbol occurrences of the right side. The transitive closure of this relation is computed and projected to the attributes of the left-side nonterminal of p. If production p is applied at the root of a parse trees t, and if the die relations $U_1, \ldots, U_k$ are the lower dependence relations for the subtrees under the root of t the lower characteristic dependence relation for t is obtained by

$$U_t(X) = [\![p]\!]^\sharp(U_1, \ldots, U_k)$$

The sets $\mathcal{U}(X), X \in V$ of all lower dependence relations for nonterminal symbols X result as the least solution of the system of equations

$$\begin{array}{lll} (\mathcal{U}) & \mathcal{U}(a) = \{\emptyset\}\,, & a \in V_T \\ & \mathcal{U}(X) = \{[\![p]\!]^\sharp(U_1, \ldots, U_k) \mid p : X \to X_1 \ldots X_k \in P, U_i \in \mathcal{U}(X_i)\}\,, & X \in V_N \end{array}$$

Here, V_T, V_N, and P are the sets of terminal and nonterminal symbols, and productions of the underlying context-free grammar. Each right side of these equations is *monotonic* in each unknown $\mathcal{U}(X_i)$ on which it depends. The set of all transitive binary relations over a finite set is finite. Therefore also the set of its subsets is finite. The least solution of this system of equations, i.e., the set of all lower dependence relations for each X can be determined by iteration. The resulting relations can be checked for cyclic dependences. Hence the non-circularity problem for attribute grammars can be decided.

Theorem 4.3.1 It is decidable whether an attribute grammar is well-formed. □

To decide well-formedness all lower dependence relations of the attribute grammar are computed. This set is finite, but it can grow exponentially in the number of attributes. The check for non-circularity is thus only practically feasible if the number of attributes is small, or if the symbols have only few lower dependence relations. In general the exponential effort is unavoidable since the problem to check for non-circularity of an attribute grammar is *EXPTIME* complete.

In many attribute grammars a nonterminal X may have several lower characteristic dependence relations, but these are all contained in one common transitive acyclic dependence relation.

Example 4.3.4 Consider the attribute grammar AG_{scopes} in Example 4.2.4. For nonterminal $block$ there are the following lower characteristic dependence relations:

$$\begin{array}{ll} (1) & \emptyset \\ (2) & \{(same, ok)\} \\ (3) & \{(env, ok)\} \\ (4) & \{(same, ok), (env, ok)\} \end{array}$$

The first three dependence relations are all contained in the fourth. □

To compute for each symbol X a transitive relation that contains all lower characteristic dependences for X we set up the following system of equations over transitive relations:

$$\begin{array}{lll} (\mathcal{R}) & \mathcal{R}(a) = \emptyset\,, & a \in V_T \\ & \mathcal{R}(X) = \bigsqcup\{[\![p]\!]^\sharp(\mathcal{R}(X_1), \ldots, \mathcal{R}(X_k)) \mid p : X \to X_1 \ldots X_k \in P\}\,, & X \in V_N \end{array}$$

The partial order on transitive relations is the subset relation $\subseteq$. Note that the least upper bound of the transitive relations $R \in \mathcal{S}$ is not just their union. Instead we have:

$$\bigsqcup \mathcal{S} = (\bigcup \mathcal{S})^+$$

i.e., following the union of the relations the transitive closure must be computed. For each production p the operation $[\![p]\!]^\sharp$ is monotonic in each of their arguments. Therefore the system of equations possesses a least solution. Since there are only finitely many transitive relations over the set of attributes this solution can be determined by iteration. Let $\mathcal{U}(X), X \in V$, and $\mathcal{R}(X), X \in V$, be the least solutions of the systems of equations $(\mathcal{U})$ and $(\mathcal{R})$. By induction over the iterations of the fixed-point algorithm One can prove that for all $X \in V$ holds

$$\mathcal{R}(X) \supseteq \bigcup \mathcal{U}(X)$$

We conclude that all characteristic lower dependence relations of the attribute grammar are acyclic if all relations $\mathcal{R}(X), X \in V$, are acyclic. An attribute grammar where all relations $\mathcal{R}(X), X \in V$, are acyclic is called *absolutely non-circular*. Each absolutely non-circular attribute grammar is thereby also well-formed. This means that for absolutely non-circular attribute grammars the algorithm for demand-driven attribute evaluation always terminates. By solving the system of equations $(\mathcal{R})$ one has identified a polynomial criterion to guarantee the applicability of demand-driven attribute evaluation.

Similar to the *lower* characteristic dependence relations of a symbol X the *upper* characteristic dependence relation for X can be defined. It is derived from attribute dependences of upper tree fragments for X. Remember: The upper tree fragment of a parse tree t at n is the tree that one obtains by replacing the subtree at n by the node n. This upper tree fragment is denoted by $t\backslash n$. Let $Dt(t\backslash n)$ be the individual dependence relation of the upper tree fragment, i.e., the set of all pairs $(n_1.a, n_2, b)$ of the individual dependence relation $Dt(t)$ for which n_1 as well as n_2 lie in the upper tree fragment $t\backslash n$. The upper characteristic dependence relation $O_{t,n}(X)$ for X at node n in t consists of all pairs $(a, b) \in \mathcal{A}(X) \times \mathcal{A}(X)$, for which the pair $(n.a, n.b)$ lies in the transitive closure of $Dt(t\backslash n)$ (see Figure 4.14). One can construct a system of equations over sets of transitive relations for the set $\mathcal{O}(X)$ of all possible upper characteristic dependence relations of symbol X (see Exercise **??**).

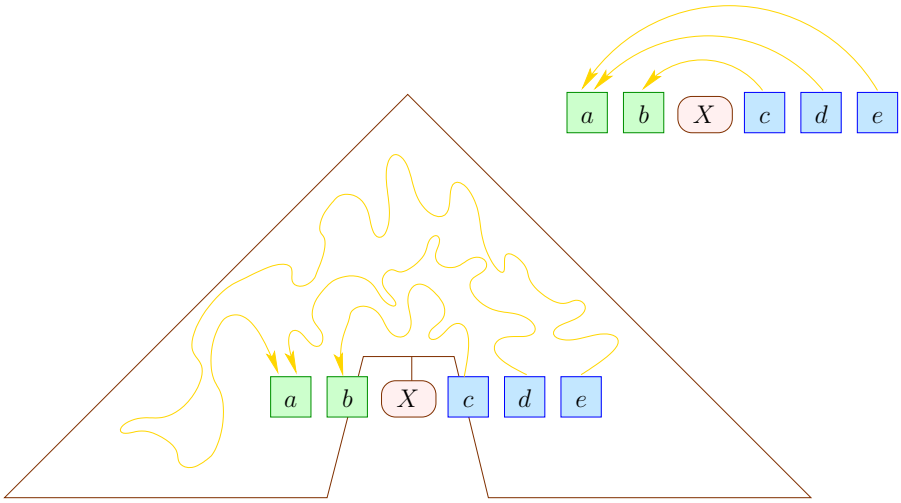


Fig. 4.14. Attribute dependences in an upper tree fragment for X and the induced upper characteristic dependence relation

4.3.3 Visit-Oriented Attribute Evaluation

The advantage of a statically generated attribute evaluator over the demand-driven dynamic evaluator of the previous section is that the behavior of the evaluator at each node is already statically fixed at generation time. No test at each attribute instance at evaluation time whether the instance is already evaluated is needed. The largest class of attribute grammars for which we describe the generation

of attribute evaluators is the class of *l-ordered* or *simple-multi-visit* attribute grammars. An attribute grammar is called *l-ordered* if there exists a function $\mathcal{T}$ that map each symbol X to a *total* order $\mathcal{T}(X) \subseteq \mathcal{A}^2$ on the set $\mathcal{A}$ of attributes of X that is compatible with all productions. This means that for each production $p: X_0 \longrightarrow X_1 \ldots X_k$ of the underlying grammar the relation

$$D_{\mathcal{T}}(p) = (Dp(p) \cup \mathcal{T}(X_0)[p,0] \cup \ldots \cup \mathcal{T}(X_k)[p,k])^+$$

is acyclic. This property is equivalent to the property that

$$\mathcal{T}(X_i) = \pi_i((Dp(p) \cup \mathcal{T}(X_0)[p,0] \cup \ldots \cup \mathcal{T}(X_k)[p,k])^+)$$

holds for all i. Therefore holds in particular:

$$\mathcal{T}(X_0) \supseteq [\![p]\!]^\sharp(\mathcal{T}(X_1), \ldots, \mathcal{T}(X_k))$$

By comparing this inequality with the equation for the unknown X_0 in the system of equations $(\mathcal{R})$ in the last section, we can conclude that the total order $\mathcal{T}(X_0)$ contains the dependence relation $\mathcal{R}(X_0)$. Since $\mathcal{T}(X_0)$ is a total order and is therefore acyclic the attribute grammar is absolutely noncircular, where all local lower dependence relations at X_0 are contained in $\mathcal{T}(X_0)$. In analogy, one can show that $\mathcal{T}(X_0)$ contains all *upper* dependence relations at X_0.

Example 4.3.5 (Continuation of Example 4.2.4) In the attribute grammar AG_{scopes} the following total orders on the sets of attributes offer themselves for the symbols $stat$, $block$, $decl$, E and var:

$stat$	$env \rightarrow ok$
$block$	$same \rightarrow env \rightarrow ok$
$decl$	new
E	$env \rightarrow ok$
var	id

□

Let $B_{\mathcal{T}}(X) \in \mathcal{A}(X)^*$ be the sequence of the attributes of X according to the total order $\mathcal{T}(X)$. This linear sequence can be factored into subsequences of inherited and purely synthesized attributes. In our example this factorization is for all considered symbols very simple: All inherited attributes occur before all synthesized attributes. In general some inherited attributes can depend on synthesized attributes. We then obtain a factorization:

$$B_{\mathcal{T}}(X) = I_{X,1}S_{X,1} \ldots I_{X,r_X}S_{X,r_X}$$

where $I_{X,i} \in \mathcal{I}(X)^*$ and $S_{X,i} \in \mathcal{S}(X)^*$ holds for all $i = 1, \ldots, r_X$ and furthermore $I_{X,i} \neq \epsilon$ for $i = 2, \ldots, r_X$ and $S_{X,i} \neq \epsilon$ for $i = 1, \ldots, r_X - 1$.

Intuitively, this factorization of the sequence $B_{\mathcal{T}}(X)$ means That the synthesized attributes at each node of a parse tree labeled with X can be evaluated in at most r_X visits; at the first visit of the node, coming from the parent node, the values of the inherited attributes in $I_{X,1}$ are available, at the return to the parent node, the values of the synthesized attributes in $S_{X,1}$ are evaluated. Correspondingly, at the ith visit of the node, the values of the inherited attributes in $I_{X,1} \ldots I_{X,i}$ are available, and the synthesized attributes in $S_{X,i}$ are computed. A subsequence $I_{X,i}S_{X,i}$ of $B_{\mathcal{T}}(X)$ is called a *visit* of X. To determine which evaluations may be performed during the ith visit at a node n and at the successors of the node n, one considers the dependence relation $D_{\mathcal{T}}(p)$ for the production $X_0 \longrightarrow X_1 \ldots X_k$ that is applied at n. Since the relation $D_{\mathcal{T}}(p)$ is acyclic $D_{\mathcal{T}}(p)$ can be arranged into a linear order. In our case we chose the order $B_{\mathcal{T}}(p)$, which can be factorized into *visits*. Altogether we obtain for the relation $D_{\mathcal{T}}(p)$ a visit sequence:

$$B_{\mathcal{T}}(p) = B_{\mathcal{T},1}(p) \ldots B_{\mathcal{T},r_{X_0}}(p)$$

The ith subsequence $B_{\mathcal{T},i}(p)$ describes what happens during the ith visit of a node n at which the production $p: X_0 \longrightarrow X_1 \ldots X_k$ has been applied. For each occurrence of inherited attributes of the

X_j ($j > 0$) in the subsequence, the corresponding attribute instances are computed one after the other. After the computation of the listed inherited attribute instances of the i'th visit of the jth successor this successor is recursively visited to determine the values of the synthesized attributes, associated with the i'th visit. When the values of the synthesized attributes of all successors are available that are directly or indirectly needed for th computation of the synthesized attributes of the ith visit of the left side X_0 the values of these synthesized attributes are computed.

To describe the subsequence $B_{\mathcal{T},i}(p)$ in an elegant way we introduce the following abbreviations. Let $w = a_1 \dots a_l$ be a sequence of attributes of the nonterminal X_j. $p[j].w = p[j].a_1 \dots p[j].a_l$ shall denote the associated sequence of attribute ccurrences in p. The i'th visit $I_{X_j,i'} S_{X_j,i'}$ of the jth symbol of the production p is denoted by the sequence $p[j].(I_{X_j,i'}\ S_{X_j,i'})$. The sequence $B_{\mathcal{T},i}(p)$, interpreted as a sequence of attribute occurrences in p, has then the form:

$$\begin{aligned} B_{\mathcal{T},i}(p) = {} & p[0].I_{X_0,i} \\ & \quad p[j_1].(I_{X_{j_1},i_1}\ S_{X_{j_1},i_1}) \\ & \quad \dots \\ & \quad p[j_r].(I_{X_{j_r},i_r}\ S_{X_{j_r},i_r}) \\ & p[0].S_{X_0,i} \end{aligned}$$

for an appropriate sequence of pairs $(j_1, i_1), \dots, (j_r, i_r)$. It consists of the visits of the nonterminal occurrences $X_{j_1}, \dots, X_{j_r}$ of the right side of production p that are embedded in the ith visit of the left side of p.

Let p be a production and $f(p[j_1].a_1, \dots, p[j_r].a_r)$ be the right side of the semantic rule for the attribute occurrence $p[j].a$ for a total function f. For a node n in the parse tree at which production p has been applied, we define

$$\mathsf{eval}_{p,j,a}\ n = f(n[j_1].a_1, \dots, n[j_r].a_r)$$

The functions $\mathsf{eval}_{p,j,a}$ are used to generate a function $\mathsf{solve}_{p,i}$ from the ith subsequence $B_{\mathcal{T},i}(p)$ of production p:

$$\begin{aligned} \mathsf{solve}_{p,i}\ n = {} & \mathbf{forall}\ (a \in I_{X_{j_1},i_1}) \\ & \quad n[j_1].a \leftarrow \mathsf{eval}_{p,j_1,a}\ n; \\ & \mathsf{visit}_{i_1}\ n[j_1]; \\ & \dots \\ & \mathbf{forall}\ (a \in I_{X_{j_r},i_r}) \\ & \quad n[j_r].a \leftarrow \mathsf{eval}_{p,j_r,a}\ n; \\ & \mathsf{visit}_{i_r}\ n[j_r]; \\ & \mathbf{forall}\ (a \in S_{X_0,i}) \\ & \quad n.a \leftarrow \mathsf{eval}_{p,0,a}\ n; \end{aligned}$$

Example 4.3.6 (Continuation of Example 4.2.4) The production-local dependence relation for the production $block \rightarrow decl\ block$ is obtained from the relation in Figure 4.8 by associating the total order on the attribute occurrences with the symbols. Altogether, this relation is embedded into the following total order:

$$\begin{aligned} & block[0].same \rightarrow block[0].env \rightarrow \\ & \quad decl.new \rightarrow \\ & \quad block[1].same \rightarrow block[1].env \rightarrow block[1].ok \rightarrow \\ & block[0].ok \end{aligned}$$

According to this total order, the evaluator first descends ino the subtree for the nonterminal $decl$ to determine the value of the attribute new. Thereby the inherited attributes of the nonterminal $block$ don the right side of the production can be computed. A descent into the subtree for this nonterminal permits to compute the value of the synthesized attribute ok of this nonterminal. After this, all values are available that are needed to compute the value of the synthesized attribute ok on the left side of the production. □

The evaluation orders in visit_i are chosen in such a way that the value of each attribute instance $n[j'].b$ will be computed before any attempt to read its value. The functions $\mathsf{solve}_{p,i}$ are simultaneously recursive with themselves and with the functions visit_i. For a node n let $\mathsf{get\_prod}\ n$ be the production that was applied at n or Null if n is a leaf that is labeled with a terminal symbol or ϵ. If $p_1, \ldots, p_m$ is a sequence of the productions of the grammar, Then the function visit_i is given by:

$$\begin{aligned}
\mathsf{visit}_i\ n \quad &= \mathbf{match}\ \mathsf{get_prod}\ n \\
&\quad\ \mathbf{with}\ \mathsf{Null} \rightarrow\ () \\
&\quad\ \mid \quad p_1 \ \rightarrow\ \mathsf{solve}_{p_1,i}\ n \\
&\qquad\qquad \ldots \\
&\quad\ \mid \quad p_m \ \rightarrow\ \mathsf{solve}_{p_m,i}\ n
\end{aligned}$$

For a node n the function visit_i checks wether n is a leaf or wether it was generated by the application of a production. If n is a leaf the evaluator doesn't need to do anything, if we assume that the synthesized attributes of the leaves were properly initialized. The evaluator recognizes that n is a leaf if the call $\mathsf{get\_prod}\ n$ returns the value Null. If n is not a leaf the call $\mathsf{get\_prod}\ n$ returns the production p_j at n. In this case thefunction $\mathsf{solve}_{p_j,i}$ is called for n.

Let S be the start symbol of the context-free grammar underlying the attribute grammar. If S has no inherited attributes then $B_{\mathcal{T}}(X)$ consists of one order of only synthesized attributes, which one can evaluate in a single visit. The evaluation of all attribute instances in a parse tree t with root n_0 for the start symbol S is then performed by the call $\mathsf{visit}_1\ n_0$.

The evaluator just presented can be generated in polynomial time from the attribute grammar together with the total orders $\mathcal{T}(X), X \in V$. Not every attribute grammar possess such a system compatble total orders. The question wether an attribute grammar is l-attributed is certainly in NP, since total orders $\mathcal{T}(X), X \in V$, can be guessed and then checked for compatibility in polynomial time. A significantly better algorithm is not known. This problem is not only in NP, but it is NP complete.

In practice one will therefore use only a subclass of the l ordered attribute grammars where a simple method delivers compatible total orders $\mathcal{T}(X), X \in V$. The starting point for the construction is the system of equations:

$$\begin{aligned}
(\mathcal{R}') \qquad \mathcal{R}'(X) \ = \ \bigsqcup \{ & \pi_i((Dp(p) \cup \mathcal{R}'(X_0)[p,0] \cup \ldots \cup \mathcal{R}'(X_k)[p,k])^+) \mid \\
& p:\ X_0 \rightarrow X_1 \ldots X_k \in P, \quad X = X_i \} \ , \qquad X \in V
\end{aligned}$$

over the *transitive* relations on attributes, ordered by the subset relation $\subseteq$. Bear in mind that the least upper bound of transitive relations $R \in \mathcal{S}$ is given by:

$$\bigsqcup \mathcal{S} = (\bigcup \mathcal{S})^+$$

The least solution of the system of equation $(\mathcal{R}')$ exists since the operators on the right side of the equations are monotonic. The least solution can be determined by the iterative method that we used in Chapter 3.2.5 for the computation of the first_ksets. Termination is guaranteed since the number of possible transitive relations is finite.

Let $\mathcal{R}'(X)$, for $X \in V$ be the least solution of the system of equations. Each system $\mathcal{T}(X), X \in V$, of compatible *total* orders is a solution of the system of equations $(\mathcal{R}')$. Therefore $\mathcal{R}'(X) \subseteq \mathcal{T}(X)$ holds for all symbols $X \in V$. If there exists such a system $\mathcal{T}(X), X \in V$, of compatible total orders the relations $\mathcal{R}'(X)$ are all acyclic. The relations $\mathcal{R}'(X)$ are therefore a good starting point to construct total orders $\mathcal{T}(X)$.

This construction is done in a way that for each X a sequence with a minimal number of visits is obtained. For a symbol X with $\mathcal{A}(X) \neq \emptyset$ a sequence $I_1 S_1 \ldots I_r S_r$ is computed, where I_i and S_i are sequences of inherited and synthesized attributes, respecively. All already listed attributes are collected in a set D, which is initialized with the empty set. Let us assume, $I_1, S_1, \ldots I_{i-1}, S_{i-1}$ are already computed, and D would contain all attributes that occur in these sequences. Two steps are executed:

1. First, a maximally large set of *inherited* attributes of X is determined that are not in D, and which only depend on each other or on attributes in D. This set is topologically sorted, delivering some sequence I_i. This set is added to D.
2. Next, a maximally large set of *synthesized* attributes is determined that are not in D, and that only depend on each other or on attributes in D. This set is added to D, and a topologically sorted sequence is produced as S_i.

This procedure is iterated producing more subsequences $I_i S_i$ until all attributes are listed, that is, until D is equal to the whole set $\mathcal{A}(X)$ of attributes of the nonterminal X.

Let $\mathcal{T}'(X), X \in V$, be total orders on the attributes of the symbols of X that are computed this way. We call the attribute grammar *ordered* if the total ordes $\mathcal{T}'(X), X \in V$, are already compatible, that is, satisfy the system of equations $(\mathcal{R}')$. In this method the relations $\mathcal{R}'(X)$ are expended *one by one* to total orders, without checking wether the added artificial dependences generate cycles in the productions. The polynomial complexity of the construction was therefore achieved with a restriction of the expressivity of the accepted attribute grammars.

In our examples 4.2.4, 4.2.3 and 4.2.5 in Section 4.2.2 attribute evaluators are generated by our method that visit each node of a parse tree exactly once. Not all practically relevant attribute grammars have so simple evaluators. An attribute grammar to compute a symbol table for JAVA, for instance, must visit the body of a class several times because in JAVA for methods need not be declared in a *forward* declaration, as is the case for functions in C. A JAVA method that is used in the body of another method, although it is declared only later. A similar problem occurs in functional languages such as OCAML, when simultaneous recursive functions are introduced (see Exercise **??**).

4.3.4 Parser-directed Attribute Evaluation

In this section we consider some classes of attribute grammars that are strongly restricted in the types of attribute dependences they admit, but quite useful in practice. The introductory example 4.2.1 belongs to one of these classes. For attribute grammars in these classes, attribute evaluation can be performed in parallel to syntax analysis and directed by the parser. attributes values are administered in a stack-like fashion either on a dedicated attribute stack or together with the parser states on the parse stack. In any case, attribute values are addressed using static relative addresses such that an efficient access is possible. The construction of the parse tree, at least for the purpose of attribute evaluation, is unnecessary. attributes-grammars from these classes are therefore interesting for the compilation of simple languages by efficient compilers.

Since attribute evaluation shall be parser directed the values of synthesized attributes at terminal symbols need to be obtained by the scanner when the symbol is passed on to the parser.

L attributesd Grammars

All parsers that we consider as possibly directing attribute evaluation process their input from left to right. This suggests that attribute dependences going from right to left are not acceptable. The first grammar class introduced, L attributed grammars, excludes exactly such dependences. This class properly contains all classes subject to parser-directed attribute evaluation. It consists of those attribute grammars in normal form where the attribute instances in each parse tree can be evaluated in one left to right traversal of the parse tree. Formally we call an attribute grammar L *attributed* (abbreviated an LAG), if for each production $p: \ X_0 \to X_1 \dots X_k$ of the underlying grammar the occurrence $p[j].b$ of an inherited attribute only depends on attribute occurrences $p[i].a$ with $i < j$. Attribute evaluation in one left to right traversal can be performed using the algorithm of Section 4.3.3, which visits each node in the parse tree only once and visits the children of a node in a fixed left to right oder. For a production $p: \ X_0 \to X_1 \dots X_k$ a functionsolve_p is generated that is defined by:

$$
\begin{aligned}
\mathsf{solve}_p\ n = \quad & \mathbf{forall}\ (a \in I_{X_1}) \\
& \quad n[1].a \leftarrow \mathsf{eval}_{p,1,a}\ n; \\
& \mathsf{visit}\ n[1]; \\
& \ldots \\
& \mathbf{forall}\ (a \in I_{X_k}) \\
& \quad n[k].a \leftarrow \mathsf{eval}_{p,k,a}\ n; \\
& \mathsf{visit}\ n[k]; \\
& \mathbf{forall}\ (a \in S_{X_0}) \\
& \quad n.a \leftarrow \mathsf{eval}_{p,0,a}\ n;
\end{aligned}
$$

Here I_X and S_X are the sets of inherited and synthesized attributes of symbol X, and the expression $\mathsf{eval}_{p,j,a}\ n$ delivers the value of the right side of the semantic rule for the attribute instance $n[j].a$. The visit of a node n is realized by the function visit:

$$
\begin{aligned}
\mathsf{visit}\ n \quad = \ & \mathbf{match}\ \mathsf{get\_prod}\ n \\
& \mathbf{with}\ \mathsf{Null} \rightarrow () \\
& \quad \mid \quad p_1 \ \rightarrow \mathsf{solve}_{p_1}\ n \\
& \qquad \ldots \\
& \quad \mid \quad p_m \ \rightarrow \mathsf{solve}_{p_m}\ n
\end{aligned}
$$

Again function $\mathsf{get\_prod}\ n$ returns the production that was applied at node n (or Null if n is a leaf). The attribute grammars AG_{scopes}, AG_{types} and AG_{bool} of Examples 4.2.4, 4.2.3 and 4.2.5 are all L attributed, where the last one is not in normal form.

LL Attributed Grammars

Let us consider the actions that are necessary for a parser-directed attribute evaluation:

- When reading a terminal symbol a: Receiving the synthesized attributes of a from the scanner;
- When expanding a nonterminal X: Evaluation of the inherited attributes of X;
- When reducing to X: Evaluation of the synthesized attributes of X;

An $LL(k)$ parser as it was described in the chapter on syntax analysis can trigger these actions at the reading of a terminal symbol, at expansion, and at reduction, respectively. An attribute grammar in normal form is called *LL attributed*, if

- it is L attributed, and
- the underlying context-free grammar is an $LL(k)$-grammar (for some $k \geq 1$).

The property of an attribute grammar to be LL attributed means that syntax analysis can be performed by an LL parser, and that whenever the LL parser expands a nonterminal, all arguments for its inherited attributes have already been computed, or at least could have been computed.

In Chapter 3.3 we described how to construct a parser to a strong $LL(k)$ grammar. This parser administered items $[A \rightarrow \alpha.\beta]$ on its stack, where the dot in the item represented that the part of the input that was derived from α was already processed. We now extend this pushdown automaton such that it manages a second stack, the *attribute stack*, whose frames are pointed to by pointers associated with items on the parse stack. The frame for the item $[A \rightarrow \alpha.\beta]$ contains the values of the inherited attributes of the left side A and the values of the synthesized attributes of the symbols in α.

Figure 4.15 visualizes the actions of the LL parser-directed attribute evaluation.

- An *expand* transition for symbol B, pushing one of its alternatives $B \rightarrow \gamma$ onto the parse stack, creates a new frame for the values of the inherited attributes of the left side B, computes their values by the associated semantic rules into this frame.

- A *shift* transition under a terminal symbol a, moving the dot over the a, extends the frame with space for the values of the synthesized attributes of a, obtains their values from the scanner, and stores these on the attribute stack.
- A *reduce* transition assumes a complete item $[B \to \gamma.]$ on top of the parse stack and the values of the synthesized attributes of γ available in the associated frame. The values of the synthesized attributes of the left side B are computed according to the semantic rules and stored in the frame for B pointed to by the item $[A \to \alpha.B\beta]$. The complete item $[B \to \gamma.]$ is removed from the parse stack, and its frame is removed from the attribute stack. The dot in $[A \to \alpha.B\beta]$ is moved over the B.

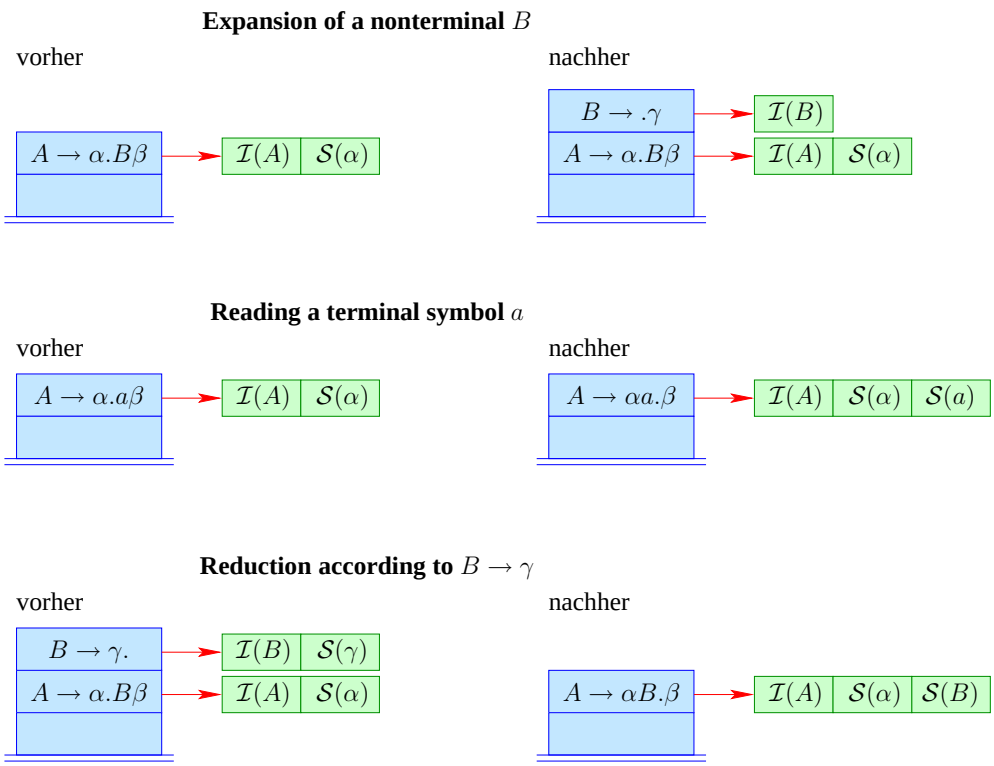


Fig. 4.15. Actions of LL parser-directed attribute evaluation, where $\mathcal{I}(A)$ and $\mathcal{S}(\alpha)$ denote the sequences of the values of the inherited attributes of a symbol A and of the synthesized attributes of the symbols in α, resp.

The attribute grammars AG_{types} and AG_{bool} are both L attributed. However, both are not LL attributed. In both cases the underlying underlying context-free grammar is left recursive and therefore not $LL(k)$ for any k. For the attribute grammar AG_{bool} one can show that there exists no LL attributed grammar that solves this code-generation problem in this way, that is, by propagation of two jump targets to each subexpression.

LR Attributed Grammars

We now present a method by which an LR parser can direct the evaluation of attributes. An LR parser administers states on its stack. States consists of sets of items, possibly extended by lookahead sets. Each such state q is associated with an attribute frame $\mathcal{S}(q)$. The attribute frame of the initial state is empty. For any other state $q \notin \{q_0, f\}$ with entry symbol X The frame $\mathcal{S}(q)$ contains the values of the synthesized attributes of the symbol X. We extend the LR parser with a (global) attribute frame $\mathcal{I}$, Which holds the value of each inherited attribute b or $\perp$ if the value of the attributes b is not available. Initially the global attribute frame $\mathcal{I}$ contains the values of the inherited attributes of the start symbol.

The values of the synthesized attributes of a terminal symbol are made available by the scanner. Two problems need to be solved if the values of the attributes for the attribute frame $\mathcal{S}(q)$ of a state q are computed:

- The semantic rule needs to be identified by which the attribute values should be evaluated.
- The values of the attribute occurrences that are arguments of the semantic rule need to accessed by static addresses.

The values of the synthesized attributes of a nonterminal X_0 can be computed when the LR parser makes a reduce transition: The production $p: \ X_0 \to X_1 \dots X_k$ is known by which the reduction to X_0 is done. To compute a synthesized attribute b of X_0 the semantic rule for the attribute occurrence $p[0].b$ of this production is used. Before the reduction a sequence $q'q_1, \dots, q_k$ of states is on top of the parse stack, where $q_1, \dots, q_k$ have entry symbols $X_1, \dots, X_k$ of the right side of p. Let us assume the values for the attribute frames $\mathcal{S}(q_1), \dots, \mathcal{S}(q_k)$ were already computed. The semantic rule for a synthesized attribute of X_0 can be applied by accessing the values for the occurrences $p[0].b$ of inherited attributes of the left side X_0 in $\mathcal{I}$ and the values for occurrences $p[j].b$ of synthesized attributes of X_j of the right side in $\mathcal{S}(q_j)$. Before the *reduce* transition, the values of the synthesized attributes of X_0 can be computed for the state $q = \delta(q', X_0)$ that is entered under X_0. Still unsolved is the question how the values of the inherited attributes of X_0 can be determined.

In the case that there are no inherited attributes, we already have a method for attribute evaluation. An attribute grammar is called S *attributed*, if it has only synthesized attributes. Example 4.2.1 is such a grammar. Despite the restriction to have only synthesized attributes once could describe how trees for expressions are constructed. In general, the computation of some semantic value can be specified by an S-grammar. This mechanism is offered by parser generators such as YACC or BISON. Each S attributed grammar is also L attributed. If an LR grammar is S attributed, the attribute frames of the states can be computed on a stack, in particular the values of synthesized attributes of the start symbol.

Attribute grammars with synthesized attributes alone are not expressive enough for more challenging tasks. Even the computation of types of expressions relative to a symbol table *env* in Example 4.2.3 requires an inherited attribute, which is passed down the parse tree. Our goal therefore is to extend the approach for S attributed grammars to deal with inherited attributes. The LR parser does in general not know the upper tree fragment, in which the transport paths for inherited attribute values lie. If a grammar is left recursive the application of an arbitrary number of semantic rules may be required to compute the value of an inherited attribute. On the other hand, it is helpful that often the values of inherited attributes passed down unchanged through the parse tree. This is the case in the attribute grammar AG_{types} of Example 4.2.3, which computes the type of an expression, in which the value of the attribute env is copied from the left side of productions in attributes of the same name to occurrences of nonterminals on the right side. This can be observed in production $block \to stat\ block$ of the attribute grammar AG_{scopes} of Example 4.2.4, in which the inherited attribute $same$ of the left side is copied to an attribute of the same name of the nonterminal occurrence $block$ of the right side, and the inherited attribute env of the left side is copied to attributes of the same name at nonterminal occurrences of the right side.

Formally we call an occurrence $p[j].b$ of an inherited attribute b at the jth symbol of a production $P: \ X_0 \to X_1 \dots X_k$ *copying* if there exists an $i < j$, such that the following holds:

1. $p[j].b = p[i].b$, and
2. $p[i].b$ is the last occurrence of the attribute b before $p[j].b$, that is, $b \notin \mathcal{A}(X_{i'})$ for all $i < i' < j$.

In this sense all occurrences of the inherited attributes env on the right side of the attribute grammar AG_{types} are copying. The same holds for the occurrences of the inherited attributes $same$ and env of the attribute grammar AG_{scopes} in the production $block \to stat\ block$.

Let us assume for a moment that all occurrences of inherited attributes in right sides were copying. This means that the values of inherited attributes do never change. If the global attribute frame $\mathcal{I}$ contains the right value of an inherited attribute, this doesn't need to be changed.

Not all occurrences of inheritedr attributes of an L attributed grammar are in general copying. For a noncopying occurrence $p[j].b$ of an inherited attribute b the attribute evaluator needs to know the production $p: \ X_0 \to X_1 \dots X_k$ and the position j in the right side of p, to select the correct

semantic rule for the attribute occurrence. We use a trick to accomplish this. A new nonterminal $N_{p,j}$ is introduced with the only production $N_{p,j} \to \epsilon$. This nonterminal $N_{p,j}$ is inserted before the symbol X_j in the right side of p. The nonterminal symbol $N_{p,j}$ is associated with all inherited attributes b of X_j that are noncopying in p. Each attribute b of $N_{p,j}$ gets a semantic rules that computes the same value as the semantic rule for $p[j].b$.

Example 4.3.7 Consider the production $block \to decl\ \ block$ of the attribute grammar AG_{scopes} of Example 4.2.4. The attribute occurrences $block[1].same$ and $block[1].env$ on the right side of the production are not copying. Therefore a new nonterminal l N is inserted before $block$:

$$\begin{aligned} block &\longrightarrow decl\ N\ block \\ N \quad &\longrightarrow \epsilon \end{aligned}$$

The new nonterminal symbol N has inherited attributes $\{same, env\}$. It doesn't need any synthesized attributes. The new semantic rules for the transformed production:

$$\begin{aligned} N.same \quad &= \mathbf{let}\ (x,\tau) = decl.new \\ &\quad\ \mathbf{in}\ block[0].same \cup \{x\} \\ N.env \quad &= \mathbf{let}\ (x,\tau) = decl.new \\ &\quad\ \mathbf{in}\ block[0].env \oplus \{x \mapsto \tau\} \\ block[1].same &= N.same \\ block[1].env &= N.env \\ block[1].ok \quad &= \mathbf{let}\ (x,\tau) = decl.new \\ &\quad\ \mathbf{in\ if}\ x \notin block[0].same \\ &\quad\quad \mathbf{then}\ block[1].ok \\ &\quad\quad \mathbf{else}\ \mathsf{false} \end{aligned}$$

Since N has only inherited attributes, it doesn't need any semantic rules. We observe that the inherited attributes $same$ and env of the nonterminal $block$ are both copying after the transformation. □

Inserting nonterminal $N_{p,j}$ doesn't change the accepted language. However the $LR(k)$ property may be lost. In Example 4.3.7 this is not the case. If the underlying context-free grammar is still $LR(k)$-grammar after the transformation we call the attribute grammar LR *attributed*.

After the transformation the only noncopying inherited attribute occurrences are the ones at the newly introduced nonterminals $N_{p,j}$. At a *reduce* transition for $N_{p,j}$ the LR parser has identified the production p and the position j in the right side of p at which it is just positioned. At reduction the new value for the inherited attribute b is computed and stored in the global attribute frame $\mathcal{I}$. The states q' which the parser may reach by a transition under nonterminal $N_{p,j}$, are associated with an attribute frame $old(q')$. This attribute frame does not contain the values of synthesized attributes. Instead the previous values of inherited attributes of $\mathcal{I}$ are stored that were overwritten by the reduction. These previous values are required to reconstruct the original values of the inherited attributes before the descend into the subtree for X.

Let us consider more precisely how the value of an inherited attribute b of the nonterminal $N_{p,j}$ can be computed. Let $\bar{p}: \ X \to \alpha.N_{p,j}\beta$ be the production that results from the transformation applied to p, where α has length m. Before the *reduce* transition for $N_{p,j}$ there is a sequence $q'q_1 \ldots q_m$ on top of the parse stack, where $q_1, \ldots, q_m$ have the sequence of entry symbols in α. The evaluation of the semantic rules for the inherited attribute b of $N_{p,j}$ accesses the values of the synthesized attributes of the symbols in α in the attribute frames of the states $q_1, \ldots, q_m$. The attribute evaluator can access the values of the inherited attribute a of the left side X in the global frame $\mathcal{I}$ if the attribute a was not defined by any $N_{p,i}$ with $i < j$ in the evaluation so far of the production $\bar{p}$. However if that was the case, the value of a can be accessed in the frame $old(q_{i'})$ to state $q_{i'}$, that corresponds to the first redefinition of a in the right side of p.

Let us consider in detail, what happens at *reduce* transition for a transformed production $\bar{p}$. Let $N_{p,j_1}, \ldots, N_{p,j_r}$ be the sequence of new nonterminals that were inserted by the transformation in

the right side of the production p, and let m be the length of the transformed right side. Before the *reduce* transition there is a sequence $q'q_1 \dots q_m$ of states on top of the parse stack where the states $q_{j_1}, \dots, q_{j_r+r-1}$ corresponf to the nonterminals $N_{p,j_1}, \dots, N_{p,j_r}$. Using the attribute frames $old(q_{j_1}), \dots, old(q_{j_r+r-1})$ the allocation of the inherited attributes before the descent into the parse tree for X is reconstructed. If an attribute b occurs in no frame $old(q_{j_i+i-1})$, $\mathcal{I}$ contains the value of b. Otherwise the value of b is set to the value of b in the first frame $old(q_{j_i+i-1})$ in which b occurs. This reconstruction of the global frame $\mathcal{I}$ for the inherited attributes is shown in Figure 4.16. If the frame $\mathcal{I}$ is reconstructed before processing the right side of production $\bar{p}$, the semantic rules for the computation of the synthesized attributes of the left side X can be applied. Any required synthesized attribute of the ith symbol occurrence of the right side of $\bar{p}$ can be accessed in the attribute frame of q_i.

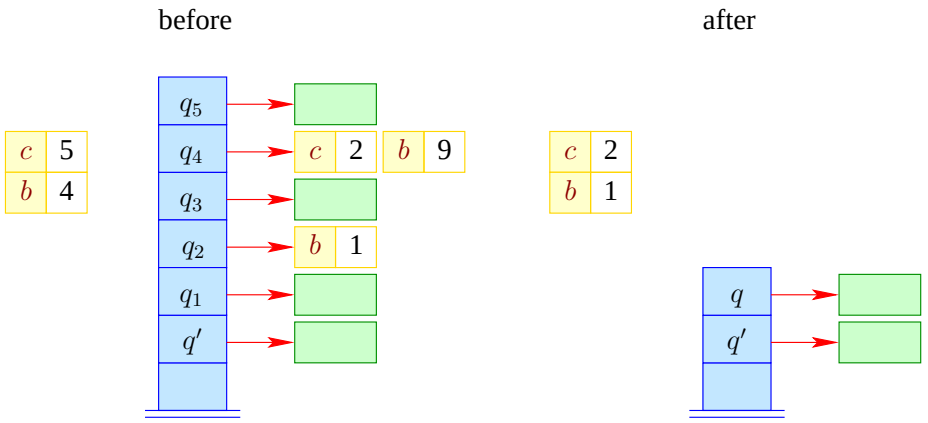


Fig. 4.16. The reconstruction of the inherited attributes at a *reduce* transition for a production $X \to \gamma$ with $|\gamma| = 5$ and $\delta(q', X) = q$. The attribute frames $old(q_2)$ and $old(q_4)$ contain the overwritten inherited attributes b and c in $\mathcal{I}$.

The described method allows extending LR parsers, such that they do not only evaluate synthesized attributes on a stack, but for LR attributed grammars also the needed inherited attributes for each state.

Example 4.3.8 The attribute grammar *BoolExp* of Example 4.2.5 is L attributed, but neither LL- nor LR attributed. IA new ε-nonterminal must be inserted at the beginning of the right side of the left recursive production for the nonterminal E since the inherited attribute $fsucc$ of nonterminal E of the right side is noncopying. Correspondingly, a new ε-nonterminal must be inserted at the beginning of the right side of left recursive production for the nonterminal T since the inherited attribute $tsucc$ of the nonterminal T is noncopying. This leaves the grammar no longer $LR(k)$ for any $k \geq 0$. □

4.4 "Ubungen

1. **1.1**
   Wie ist der Inhalt der Symboltabelle im Rumpf der Prozedur q hinter der Deklaration der Prozedur r in Beispiel 4.1.2?
2. **1.2**
   Gegeben seien die folgenden Operatoren: