4

Semantic Analysis

4.1 The Task of Semantic Analysis

Some required properties of programs cannot be described by context-free grammars. These properties are described by *context conditions*. They are meant to prevent programming mistakes. The most important examples for such context-conditions are the requirements to declare identifiers and type consistency.

Fundamental for these requirements are the rules of the programming language for *validity* and *visibility* of identifiers. The rules for *validity* determine for identifiers declared in the program what the *scope* of a declaration is, that is, in which part of the program, the declaration may have an effect. The rules for *visibility* determine, where in its scope an identifier is *visible* or *hidden*.

The rules concerning *declaredness* determine, whether an explicit declaration has to be given for an identifier, where it has to be placed, and whether multiple declarations of an identifier are allowed. The *type* of a value determines which operations can be applied to it, the type of a variable, values of which type it may take on. The *type consistency* of a program guarantees that at run time no operation can be applied to operands of the wrong type.

Some Notions

We use the following notions to describe the task of semantic analysis.

An *identifier* is a symbol (in the sense of lexical analysis), which can be used in a program to name a program element. Program elements in imperative languages that may be named are constants, variables, types, modules, functions, procedures, parameters, and statement labels. In object-oriented languages such as JAVA also classes and interfaces, their attributes and their methods, and their types can be named. In functional languages such as OCAML variables and functions can be named by identifiers. Data structures can be built using constructors, another class of identifiers. The concept of these constructors can be seen as a generalization of enumeration types in imperative languages, which list a sequence of constants whose identifiers are introduced together with the type declaration. In logic languages such as PROLOG there exist identifiers for predicates, constants, data constructors, and variables.

Some identifier are introduced in explicit declarations. The occurrence of an identifier in its declaration is the *defining occurrence* of the identifier, all other occurrences are *applied occurrences*. In imperative programming languages such as C and in object-oriented languages such as JAVA all identifiers need to be explicitly introduced. This also holds essentially for functional languages such as OCAML. In PROLOG, however, neither the used constructors and atoms nor the local variables in clauses are explicitly introduced. To make a difference between them the different kinds of identifiers are taken from different name spaces. Variables begin with a capital letter or an underscore, constructors and atoms with lower-case letters. The term $f(X, a)$ represents an application of the binary constructor $f/2$ to the variable X and the atom a. Instead of through an explicit declaration an identifier is introduced by its syntactically first occurrence in a clause.

Each programming language has *scoping constructs*, which form boundaries for where an identifier can be used. Imperative languages offer blocks, packages, modules, function and procedure declarations for this purpose. Object-oriented languages such as JAVA additionally have classes and interfaces, which may be organized in hierarchies. Functional languages such as OCAML also offer modules to collect a set of declarations. Declarations of variables and functions allow to restrict the use of identifiers to program parts. PROLOG essentially has only the clause as structuring concept.

Such structuring concepts are called *scope* constructs. A method declaration in a JAVA program is a scope in the same way as a module in OCAML, or a clause in PROLOG.

The *type* of a program element restricts what can be done with the element during the execution of the program. A value of type **int** can only be operated on by arithmetic operations together with other values of type **int**. These operations yield again values of type **int**. *int* values are internally represented in a fixed way, for instance, as 32 Bits and need therefore always the same space. The compiler for an imperative language reserves this space for variables of type **int**. It can rely on the fact that at run time any access to this allocated memory will always find an **int** value. In purely functional languages *int* values cannot be explicitly stored. A variable denotes an expression, which can be reduced to a value. The type of a variable must therefore match the types of the values that the variable may have.

Concrete and Abstract Syntax

Conceptually, the input to semantic analysis is some representation of the structure of the program as it is produced by syntactic analysis. The most popular representation of this structure is the parse tree according to the context-free for the language. This tree is called the *concrete syntax* of the program. The context-free grammar for a programming language contains information that is important for syntax analysis, but irrelevant for the subsequent compiler phases. Typical for this kind of information are the nonterminals used to express operator precedence in expressions. Once the syntactic structure of the program is recognized these nonterminals have lost their function. Therefore compilers use simplified representations of the syntactic structure, called *abstract syntax* trees. It only represents the constructs occurring in the program and their nesting.

Example 4.1.1 The concrete syntax of the program fragment

$$\begin{aligned} &\textbf{if } (x + 1 > y) \\ &\quad z \leftarrow 1; \\ &\textbf{else } z \leftarrow 2; \end{aligned}$$

is shown in Figure 4.1. We assumed that the associated context-free grammar differentiates between the precedence levels for assignments, comparison, addition, and multiplication operators. Notable are the long chains of chain reductions, that is, replacements of one nonterminal by another one, which were introduced to bridge the precedence differences. An abstract parse tree does not contain these sequences of chain reductions. For each statement type there is one constructor forming a tree out of the constituents of the statement. Similarly for arithmetic expressions; operators are used as constructors and the operands as their constituents. Figure 4.2 shows a corresponding abstract representation. □

In the following, we will sometimes use concrete, sometimes abstract syntax whatever is more intuitive. Compiler, however, always use abstract syntax for semantic analysis and for some parts of code generation.

4.1.1 Rules for Validity and Visibility

Programming languages often allow to use one identifier for different program elements and to have several declarations of the identifier. Thus a rule is needed that determines to which defining occurrence an applied occurrence refers. These rules are the ones for *validity* and for *visibility*.

The *scope* of the defining occurrences of an identifier x is that part of a program in which an applied occurrence of x can refer to this defining occurrence.

The *visibility* of a defining occurrences of an identifier can be restricted in its scope by hiding it. On the other hand there exist possibilities to make an identifier explicitly visible even outside of its

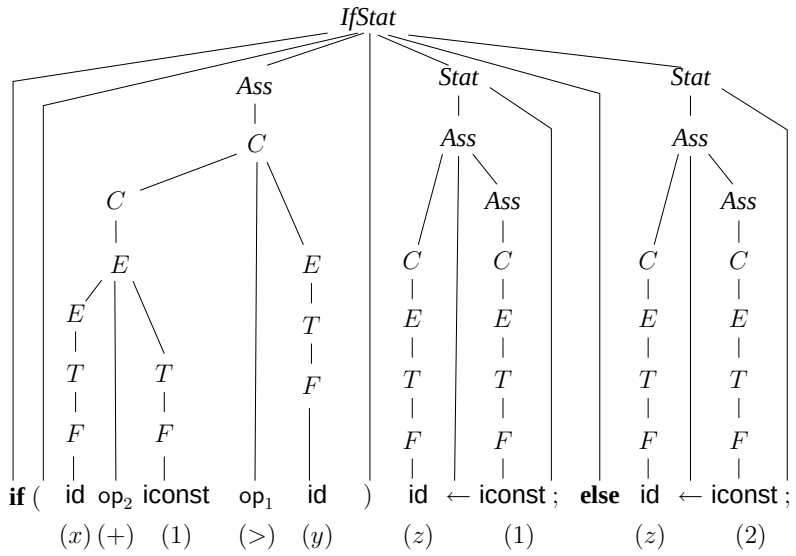


Fig. 4.1. Representation of the concrete yntax

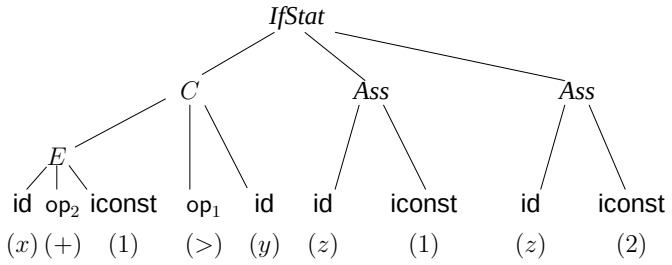


Fig. 4.2. representation of the abstract syntax

scope. These means are the *qualification* of components of structured types and the *import* of identifiers through use clauses.

The process of *identification of identifiers*. identifies for each applied occurrence of an identifier the defining occurrence that belongs to this applied occurrence according to the rules of validity and visibility of the language. We will see later that in programming languages that allow overloading of identifiers an applied occurrence of an identifiers can refer to several defining occurrences.

The validity and the visibility rules of a programming language are strongly related to the type of nesting of scopes that the language allows.

COBOL has no nesting of scopes; all identifiers are valid and visible everywhere. FORTRAN77 only allows one nesting level, that is, no nested scopes. Procedure and functions are all defined in the main program. Identifiers that are defined in a block are only visible within this block. An identifier declared in the main program is visible starting with its the declaration, but hidden within procedure declarations that contain a new declaration of the identifier.

Modern imperative and object-oriented languages such as PASCAL, ADA, C, C++, C♯ and JAVA and functional programming languages allow arbitrarily deep nesting of blocks. The ranges of validity and visibility of defining occurrences of identifiers are fixed by additional rules. In a *let* construct

$$\textbf{let } x = e_1 \textbf{ in } e_0$$

in OCAML the identifier x is only valid in the *body* e_0 of the *let* construct. Applied occurrences of x in the expression e_1 refer to defining occurrence of x in enclosing blocks. The scope of the identifier $x_1, \ldots, x_n$ of an *let-rec* construct

$$\textbf{let rec } x_1 = e_1 \textbf{ and } \ldots \textbf{ and } x_n = e_n \textbf{ in } e_0$$

consists of the expressions $e_0, e_1, \ldots, e_n$. This rules makes it difficult to translate programs in one pass; when translating a declaration the compiler may need informations about an identifier whose declaration has not yet been processed. In PASCAL, ADA and C therefore have *forward*-declarations to avoid this problem.

PROLOG has several classes of identifiers, which are characterized by their syntactic positions. There is no hiding; so validity and visibility agree. The identifiers of the different classes have the following scopes:

- Predicates, atoms, and constructors have global visibility; they are valid in the whole Prolog program and in associated queries.
- Identifiers of clause variables are valid only in the clause in which they occur.

Explicit declarations exist only for predicates: These are defined by the list of their alternatives.

Scoping in JAVA, not yet translated

Conclusion

Not everywhere in the scope of a defining occurrence of x an applied occurrence of x refers to this defining occurrence. If the defining occurrence is *global* to the actual block that is, not declared in its declaration part a local (re)declaration of x can *hide* it. It is then not *directly visible*. However, there exist several possibilities to make a not directly visible defining occurrence of an identifiers x visible within its scope. The visibility rules of a programming language determine to which defining occurrence of an identifier an applied occurrence may refer. The *extension of an identifier* by another identifier of a construct containing the declaration makes it possible to refer to a hidden defining occurrence. We have seen how complex these rules may be at the example of the language JAVA.

- Some languages have directives that allow to make a hidden defining occurrence of an identifier visible in a part of its scope, a *region*, without extension of the identifier. We have seen the *import* directive in JAVA. These directives are part of the static and not of the dynamic semantics. The boundaries of a region are determined by associated opening and closing parentheses, such as the *with* directive in PASCAL, are they are the boundaries of the directly enclosing program unit. In JAVA this is the file. The *use* directive in ADA lists identifiers of enclosing program units whose declarations are thereby made visible. The visibility of these identifiers stretches from the end of the *use* directive to the end of the enclosing program unit.
- If a language (and not only its implementation) knows the concept separate compilation of program units there exist directives to make declarations in separately compiled units visible. Each separately compiled program unit can offer some of their declarations for external reference. In JAVA this can be controlled by the modifier **public**, **protected** and **private**. In ADA a package (module) offers the definitions of its public part, individual procedures offer their formal parameters, These identifiers are again associated with scopes enclosing all programs that are linked together after being separately compled. The *with* directive in ADA makes identifiers visible that are offered by separately compiled units and mentioned in the *with* list.

In conclusion we record that the scope of the defining occurrence of an identifier, its range of validity, is the part of a program in which the identifier can be used to access the element named by the identifier. This element is either directly visible or can be made visible.

4.1.2 Checking the Context-Conditions

We will now sketch how compilers check the context-conditions. We consider the simple case of a programming language with nested scopes, but without overloading.

The task is decomposed in two subtasks. The first subtask consists in checking whether identification are declared and in identifying defining with applied occurrences. We call this task *declaration analysis*. This analysis is determined by the rules for validity and visibility of the programming language. The second subtask, *type checking*, examines whether the types of program objects confirm to the rules of the type system. It may also infer types for objects for which no types were given.

Identification of Identifiers

The rules for validity and visibility determine in our simple case that in a correct program, exactly one defining occurrence of an identifier belongs to each applied occurrence of the identifier. The identification of identifiers consists in linking each applied occurrence to a defining occurrence or to find that no such linking is possible or that more than one exist. The result of this identification is later used for type checking and possibly for code generation. It must therefore be passed on to subsequent compiler phases. There exist several possibilities for the representation of the link between applied and defining occurrence. Traditionally the compiler builds a so-called *symbol table*, in which the declarative information for each defining occurrence of an identifier is stored. This symbol table is frequently organized similarly to the block structure of the program. This helps to quickly reach the corresponding defining occurrence starting from an applied occurrence.

Such a symbol table is not the result of the identification of identifiers, but it support the identification. The result of the identification consists in the establishment of the links from the nodes for applied occurrences of an identifier x to the node of defining occurrence of x. abgespeichert ist.

Which operations must the symbol table offer? When the declaration analyzer meets a declaration it must enter into the symbol table the identifier and a reference to the declaration node in the parse tree. Another operation must register the opening of a block, yet another the closing of a block. The latter operation can delete the entries for the declarations of the closed block from the symbol table. This way, the symbol table contains at any point in time exactly the entries for declarations of all actually opened, but not yet closed blocks. When the declaration analyzer meets an applied occurrence of an identifier it searches the symbol table according to the rules for validity and visibility for the entry of the corresponding defining occurrences. When it has found this entry it copies the reference to the declaration node to the node for the applied occurrence.

Thus the following operations on the symbol table are required:

(a)	*create_symb_table*	creates an empty symbol table.
(b)	*enter_block*	registers the opening of a new block.
(c)	*exit_block*	resets the symbol table to the state before the last *enter_block*.
(d)	*enter_id*(id, *decl_ptr*)	enters an entry for identifier id into the symbol table. This contains the reference to its declaration node, which is passed in *decl_ptr*.
(e)	*search_id(*id*)*	searches the defining occurrence to id and returns the reference to the declaration node if it exists.

The last two operations (functions) work relative to the last opened block, the actual block.

Before we describe a possible implementation of the symbol table, that is, of the procedures and functions listed above we demonstrate their use by the declaration analysis. We assume validity rules as those for ADA; the defining occurrence of an identifier is only valid starting with the end of the declaration.

```
proc analyze_decl (k : node);
    proc analyze_subtrees (root: node);
    begin
      for i := 1 to #descs(root) do      (* #descs: number of children *)
          analyze_decl(root.i)           (* i-th child of root *)
      od
    end;
begin
    case symb(k) of          (* label of k *)
    block:    begin
                  enter_block;
                  analyze_subtrees(k);
                  exit_block
              end;
    decl:     begin
```

```
                    analyze_subtrees(k);
                    foreach identifier id decl. here do
                        enter_id(id, ↑ k)
                    od
                end;
        appl_id:    (* applied occurrence of identifier id *)
                    store search_id(id) an k;
        otherwise: if k no leaf then analyze_subtrees(k) fi
        od
end
```

The Ada rules for validity determine that in the *decl* case of the *case*-statement first all declarations are recursively processed before the local declarations are entered. The modification of this algorithm for ALGOL-like and PASCAL-like validity rules are left to the reader (see Exercise 1.4).

Checking Type Consistency

The check for type consistency can be performed in one bottom-up pass over the expression tree. For terminal operands, that are constants the type is fixed. For identifiers the analysis obtains the type from their declaration. For each operator one checks in a table (see Figure 4.4), whether the types of the operands match the requirements of the operator and which is the result type. If overloading is allowed, the right operation is selected for overloaded operators.

If the programming language allows type conversions, for instance, from integer $\rightarrow$ real, then it must be checked for each operator and each combination of operand types that don't match whether the operand types can be made to match by type conversions.

Implementation of the Symbol Table

The implementation of a symbol table must guarantee that the *search_id*-function finds the correct entry for an identifier according to the validity rules out of several coexisting entries.

A first solution could be a linear list of *enter_block*- and *enter_id*-entries. New entries are attached at the end, and *exit_block* erases the last *enter_id*-entries including the last *enter_block*-entry. *search_id* searches the list from the end and finds all actually valid identifiers according to the Ada validity rules. This linear lists is apparently administered in a stack-like fashion So, one can actually organizing it as a stack.

Disadvantageous at this solution is the high effort for *search_id*, which linearly depends on the number of declared identifiers. We can obtain a solution with logarithmic search time if the entries for each block are arranged in a binary search tree. The search starts with the search tree for the actual block and continues with the search tree for the enclosing block until a defining occurrence is found, or until it is clear that no such occurrence exists.

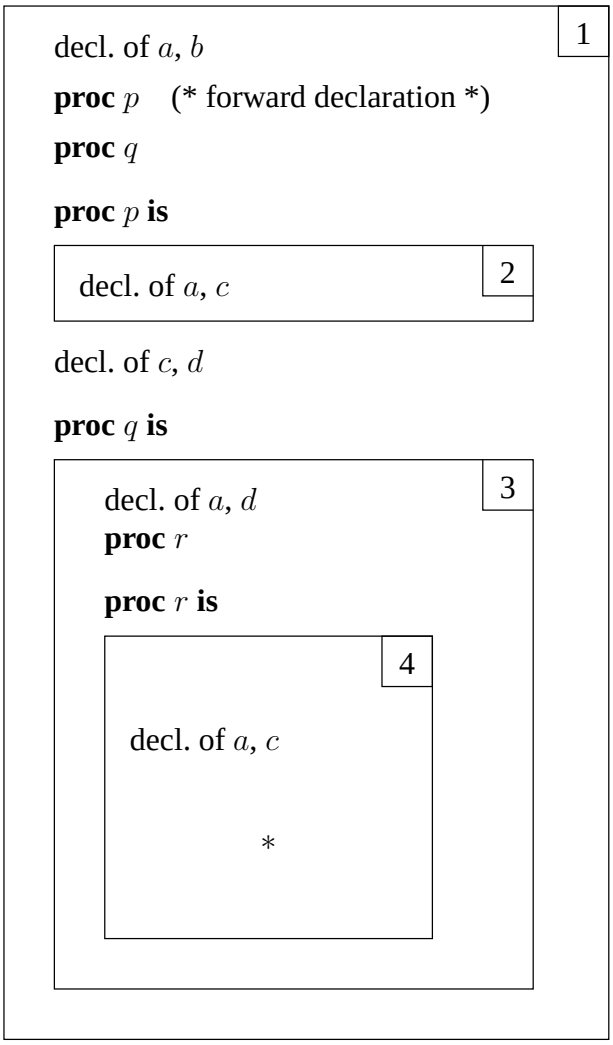
Under the assumption that each defined identifier has several applied occurrences *search_id* should be made very efficient, if possible in constant time. This can be achieved by using the following data structure:

The entries of all currently valid defining occurrences of each identifier are kept in a singly linked list; each new definition is attached to the end of this list. This last entry is pointed to by a component of an array that is indexed by the identifier. All entries belonging to the same block are also linked in a chain to support the procedure *exit_block*. A list head associated with this block points to this chain. The set of these list heads can be managed in a stack-like fashion.

Example 4.1.2
Figure 4.4 shows the symbol table for the program in Figure 4.3 and den program point labeled with $*$. □

The implementation of the symbol-table operations is as follows:

Fig. 4.3. Nested scopes

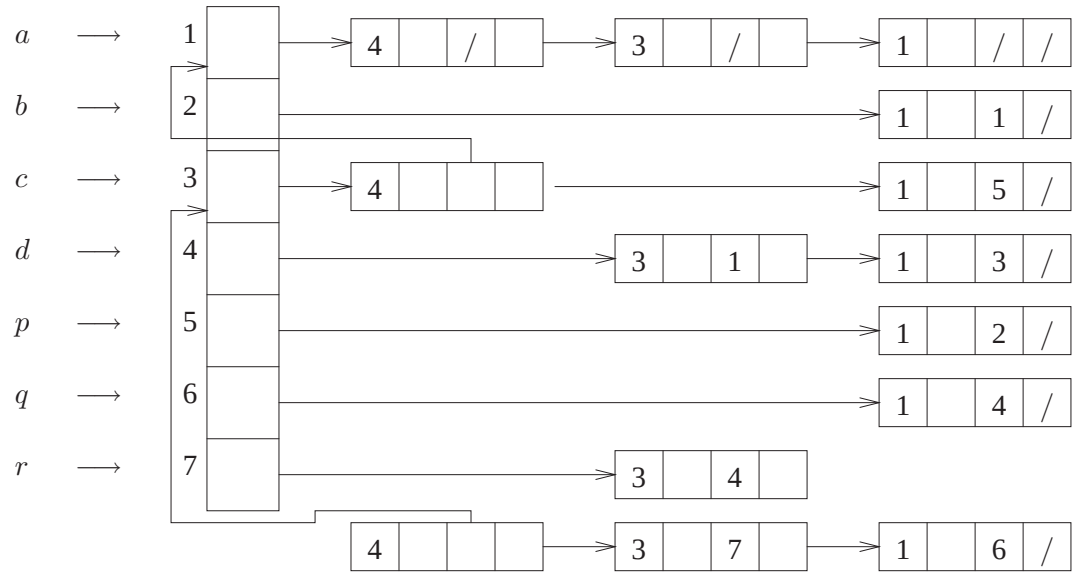



```
proc  create_symb_table;
    begin
        create empty stack of block entries
    end;

proc  enter_block;
    begin
        push entry for new block onto stack
    end;

proc  exit_block;
    begin
        foreach  declaration entry of the actual block do
            delete entry
        od;
        delete block entry from stack
    end;
```

Fig. 4.4. Symboltabelle zum Programm aus Abbildung 4.3. Die Verkettung der Eintr"age zu einem Block ist nur f"ur Block 4 angegeben, da sonst das Diagramm zu un"ubersichtlich w"urde. Sonst sind die “Adressen”der K"astchen, die Zahlen 1 - 7 angegeben. Das nicht ausgef"ullte K"astchen in jedem Eintrag enth"alt jeweils den Verweis auf den Unterbaum f"ur die Deklaration.



proc *enter_id* (*id: Idno; decl:* ↑ *node*)*;*
begin
if there exists an entry for *id* in this block
then *error*(″double declaration″)
fi*;*
create new entry with *decl* and no. of actual block;
attach this entry at the end of the linear list for *id ;*
attach this entry at the end of the linear list for this block *;*
end*;*

function *search_id* (*id: idno*) ↑ *node;*
begin
if list for *id* is empty
then *error*(″undeclared identifier″)
else return (value of decl field of first entry in list for *id*)
fi
end

4.1.3 Overloading of Identifiers

A symbol is *overloaded*, if it may have several meanings at some point in the program. Already mathematics knows overloaded symbols, for example, the arithmetic operators, which, depending on the context, may denote operations on integral, real, or complex numbers, or even operations in rings and fields. The early programming languages Fortran and Algol60 have followed the mathematical tradition and admitted overloaded operators. A type determination as presented in the last section needs to *resolve the overloading*; it should identify the right operation for an overloaded operator depending the types of its operands and possibly the expected type of the result.

Programming languages often allow overloading of use defined identifiers, such as procedure or function names. In correct programs one applied occurrence of an identifier x may correspond to several defining occurrences of x. A redeclaration of an identifier x only hides an enclosing declaration of x if both have the same type. A program is correct only if the "type environment" of each applied occurrence allows the compiler to select exactly one defining occurrence. The type environment of procedure and function calls consists of the combination of the types of the actual parameters.

The visibility rules of Ada combined with the possibility to overload symbols lead to a hardly understandable set of conflict-resolution rules for the cases where identifiers are visible or are made visible in different ways.

Example 4.1.3 (Ada program (Istvan Bach))

```
procedure BACH is
    procedure put (x: boolean) is begin null; end;
    procedure put (x: float)   is begin null; end;
    procedure put (x: integer) is begin null; end;
    package x is
       type boolean is (false, true);
       function f return boolean;                          -- (D1)
    end x;
    package body x is
       function f return boolean is begin null; end;
    end x;
    function f return float is begin null; end;            -- (D2)
    use x;
begin
    put (f);                                               -- (A1)
    A: declare
       f: integer;                                         -- (D3)
    begin
       put (f);                                            -- (A2)
       B: declare
          function f return integer is begin null; end;    -- (D4)
          begin
             put (f);                                      -- (A3)
           end B;
    end A;
end BACH;
```

The package `x` declares in its public part two new identifiers, namely the type identifier `boolean` and the function identifier `f`. These two identifiers are made potentially visible after the semicolon by the use directive `use x;` (see after (D2)). Function identifiers in Ada can be overloaded. The two declarations of `f`, at (D1) and (D2), have different „parameter profiles“ in this case different result types. they are therefore both at program point (A1) (potentially) visible.

The declaration `f: integer` in the program unit `A` (siehe (D3)) hides the outer declaration (D2) of `f`, since variable identifiers in Ada cannot be overloaded. For this reason the declaration (D1) is not visible. Declaration (D4) of `f` in program unit `B` hides declaration (D3), and since this one hides declaration (D2), transitively also D2. Declaration (D1) potentially made visible through the use directive is not hidden, but still potentially visible. In the context `put(f)` (see (A3)) `f` can only refer to declaration (D4) since the first declaration of `put` uses a type, `boolean`, that is different from the result type of `f` in (D1). □

The process of selecting the right defining occurrences of overloaded symbols is called *overload resolution*. The resolution of overloadings is performed after the identification of identifiers within certain constructs of the language, that is, restricted to expressions, (composed) identifiers etc.

The resolution algorithm works on the representation of the Ada program as abstract parse tree. Conceptually it uses four traversals of the expression tree. However, the first can be merged with the second and the third with the fourth. To formulate the algorithm we introduce some notation: At each node k of the abstract parse tree

$\#\mathit{descs}(k)$	returns the number of child nodes of k,
$\mathit{symb}(k)$	returns the symbol labeling k,
$\mathit{vis}(k)$	returns the set of definitions of *symb*(k) visible at k
$\mathit{ops}(k)$	returns the set of actual candidates for the overloaded symbol $\mathit{symb}(k)$, and
$k.i$	yields as usual the ith child of k.

For each defining occurrence of an overloaded symbol op with type $t_1 \times \cdots \times t_m \to t$ let

$$\begin{array}{lcl} \mathit{rank}(op) & = & m \\ \mathit{res\_typ}(op) & = & t \\ \mathit{par\_typ}(op,i) & = & t_i \quad (1 \le i \le m). \end{array}$$

The two latter we extend to sets of operators. For each expression in which overloading of operators needs to resolved a type, the so-called a-priori type, is computed from its context.

proc *resolve_overloading* (*root: node, a_priori_type: type*);

func *pot_res_types* (*k: node*): **set of** *type;*
(∗ potential types of the result ∗)
return $\{\mathit{res\_typ}(op) \mid op \in \mathit{ops}(k)\}$

func *act_par_types* (*k: node, i:* **integer**): **set of** *type;*
return $\{\mathit{par\_typ}(op,i) \mid op \in \mathit{ops}(k)\}$

proc *init_ops*
begin
foreach k
$\mathit{ops}(k) := \{op \mid op \in \mathit{vis}(k) \text{ and } \mathit{rank}(op) = \#\mathit{descs}(k)\}$
od;
$\mathit{ops}(\mathit{root}) := \{op \in ops(root) \mid \mathit{res\_typ}(op) = \mathit{a\_priori\_typ}\}$
end;

proc *bottom_up_elim* (*k: node*);
begin
for $i := 1$ **to** $\#\mathit{descs}(k)$ **do**
bottom_up_elim $(k.i)$;
$ops(k) := ops(k) - \{op \in ops(k) \mid \mathit{par\_typ}(op,i) \notin \mathit{pot\_res\_types}(k.i)\}$
(∗ remove the operators, whose ith parameter type does not
match the potential result types of the ith operand ∗)
od;
end;

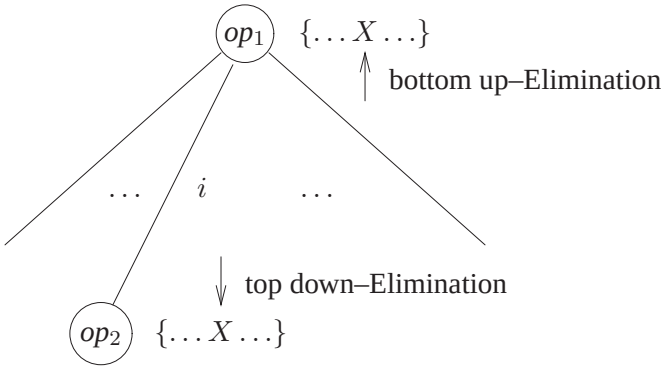
proc *top_down_elim* (*k: node*);
begin
for $i := 1$ **to** $\#\mathit{descs}(k)$ **do**
$ops(k.i) := ops(k.i) - \{op \in ops(k.i) \mid \mathit{res\_typ}(op) \notin \mathit{act\_par\_types}(k,i)\}$;
(∗ remove the operators, whose result type does not match
any type of the corresponding parameter ∗)
top_down_elim$(k.i)$
od;
end;

begin
init_ops;

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 $\mathcal{S}(X)$.

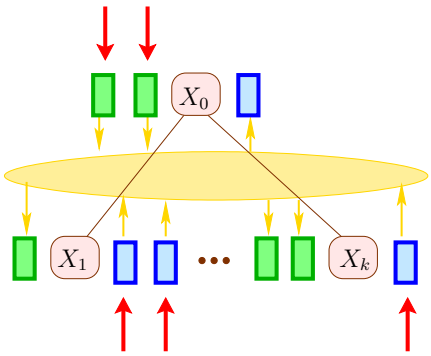


Fig. 4.6. An attributed node in the parse tree with its attributed successors. Instances of inherited attributes are drawn as boxes to the left of syntactic symbols, instances of synthesized attributes as boxes to the right of symbols. Red (darker) arrows show the information flow into the production 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.

Nat"urlich muss im Fall der identischen "Ubergabe an ein abgeleitetes attribute der linken Seite genau ein gleichnamiges abgeleitetes attribute auf der rechten Seite auftreten. Die folgenden Beispiele benutzen diese Konvention – zumindest für Regeln $A \longrightarrow \alpha X \beta$, deren rechte Seite nur ein Symbol X enthalten, dessen Attribute mit den Attributen der linken Seite A übereinstimmen.