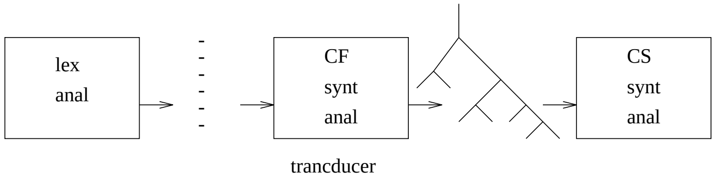
Chapter 6

SEMANTIC ACTIONS



By a **semantic action**, compiler constructors mean an action embedded in a grammar, recognizer or parser, and performed as a side effect during parsing (of the input, or of some derived internal representation). Such actions are introduced to deal with the static semantics.

In this and the following chapters we will successively consider semantic actions serving to implement the following tasks:

- storing the parse tree (which turns a recognizer into a parser)
- storing information about defining occurrences
- identification of applied occurrences (determining type, etc.)
- type equivalencing (deciding which types are equivalent and which are not)
- checking of context conditions
- production of a translation.

We will divide the syntactic analysis process into two phases, the context-free and the context-sensitive analysis.

The **CF analysis** can then be seen as a transduction from a sequence of tokens to a parse tree. In this chapter we introduce a number of strategies for producing and representing a parse tree.

6.1 Abstract Syntax Trees

Consider the following fragment of a grammar:

```
series:
        unit, semicolon symbol, series;
        unit.
```



```
unit:
        if clause;
        while clause;
        expression.

if clause:
        if symbol, series,
        then symbol, series,
        else symbol, series,
        fi symbol.

while clause:
        while symbol, series,
        do symbol, series, od symbol.
```

In principle we will store one node corresponding to each construct in the program. But obviously, if we know a node is an `if clause` we need not store the `if`-, `then`-, `else`- and `fi symbol`. They are necessary in the original, sequential representation of the program, according to the **concrete syntax** of the language but are redundant in the tree-form representation. We will avoid storing redundant symbols and nodes in the parse tree, but as a consequence the parse tree is actually built according to grammar different from the original concrete grammar. We call the resulting (nonredundant) grammar describing the syntax trees an abstract syntax for our language. The CF parser is a transducer from concrete to **abstract syntax**.

6.1.1 Abstract tree for expressions

Consider the following, already somewhat simplified syntax for expressions:

```
E --> + T | E - T | T
T --> * F | T / F | F
F --> | Var | ( E )
```

The distinction between `E`, `T` and `F` is made purely to express the different priorities of the operations – in fact all are (some form of) expressions. Parsing $a + 3 * x - b$ according to this **concrete syntax** leads to the leftmost parse tree in fig. 6.1. This concrete parse tree is a representation of the steps in the parsing process, rather than capturing "the essence" of the expression. A less redundant representation can be found in the right tree in that figure.

In fact this is a parse tree according to an **abstract grammar** for expressions, better suited than the original grammar for semantic processing.

```
E --> + E | E - E | E * E | E / E | Const | Var | ( E )
```

This abstract grammar is obtained by generalizing `F` and `T` to `E` and then eliminating `F` and `P` by substitution. (This transformation is called the *elimination of* **chain rules**). While the concrete grammar describes a string as input, the abstract grammar describes a tree. It is never used for string-parsing, and also very unsuitable because it is highly ambiguous.

6.1.2 Operator trees

A still more compact form of abstract syntax can be obtained by further eliminating redundancies, like the brackets and the explicit operators, and rearranging and renaming the

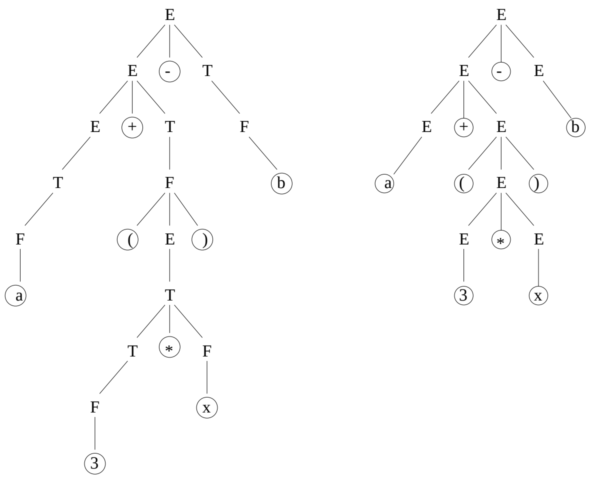


Figure 6.1: Concrete and abstract parse tree

rules. By an **operator tree** we mean an abstract syntax tree whose nodes are marked with operators (rather than nonterminal symbols) and whose edges correspond to operands.

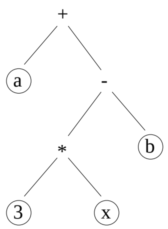


Figure 6.2: Operator tree

6.2 Explicit Trees

An **explicit tree** is a tree in which the sons of each node are explicitly pointed to by a reference in the fathernode (as opposed to an implicit tree, see 6.4.

6.2.1 Using typed references

In a language like Lalgol68, which has both strongly typed references and the union-concept, a particularly clear representation of the abstract syntax tree can be obtained as follows: we associate a different type with each nonterminal symbol (grammatical class) of the language, using unions to distinguish alternatives, references to point to members and NIL for empty alternatives.

```
TYPE SERIES      = STRUCT (UNIT hd, REF SERIES tl),
```



```
     UNIT        = UNION (IFCLAUSE, WHILECLAUSE, EXPRESSION),
     IFCLAUSE    = STRUCT (REF SERIES cond, then, else),
     WHILECLAUSE = STRUCT (REF SERIES cond, body)
```

and so on. Notice the `REF`'s.

In a recursive descent parser for the concrete language, each recognizer can now be equipped with an output parameter of the type associated with its nonterminal symbol, building up the parse tree by way of a semantic action, e.g.

```
PROC series = (REF SERIES e) VOID:
  BEGIN
    UNIT u, SERIES s := NIL;
    unit (u);
    IF is semicolon symbol
    THEN
      series (s);
    FI
    e := HEAP SERIES := (u,s)
 END;
```

and so on. In this way, simple and reliable parsers are easily obtained from LL(1) grammars, whereas the strong typing mechanism helps in guaranteeing the consistency and correctness of the semantic actions and their arguments.

Parsing the resulting abstract syntax tree involves a heavy use of conformity relations, but is also easy to get correct. The resulting parsers are somewhat slow but their main drawback is the large amount of space necessary for storing all those references (including those hidden in the UNION's).

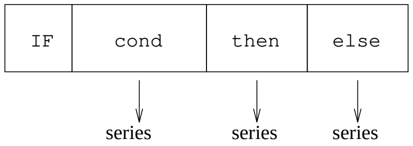
6.2.2 Using untyped references

The same idea can be expressed just as well in a typeless language, provided to each node is added a **marker** to indicate its type. If we use the first cell of each node as a marker, the type of each reference can be found by following it to its target. We have then effectively a form of dynamic type system.

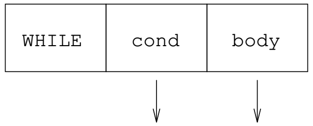
Expressing the preceding example in Lc, and compressing the abstract syntax somewhat further, we will introduce the following types of nodes:

- unit:
  An unit is either an ifclause, or an assignation, or an expression, where each node begins with a distinguishing marker

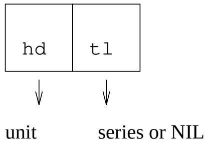
- ifclause:
  An ifclause will be represented by a unique `IF` marker and references to the nodes for the condition, the then-part and the else-part.



- whileclause:
  A whileclause will be represented by an unique `WHILE` marker and references to the nodes for the condition and the body of the while.



- series:
  A series will be represented as a linear list, without a distinguishing marker and with NIL at the end, since in all syntactic positions where a series can occur no other construct can occur.



and so on.

6.2.3 Solution in Łc

Again, recursive descent recognizers can be constructed with one output parameter to hold the resulting parse tree.

```
void should_be_series(int *v)
{
  should_be_unit(v),
  if (is_semicolon()) {
    int s;
    should_be_series(&s);
    enter_pair(*v,s,v);
  };
}

void should_be_unit(int *v)
{
  if (is_if_clause(v))
    return;
  if (is_assignation(v))
    return;
  should_be_expression(v);
}

int is_if_clause(int *v)
{
  if(is_if_symbol()) {
    int cond_series,then_series,else_series;
    should_be_series(&cond_series);
    should_be_then_symbol();
    should_be_series(&then_series);
    should_be_else_symbol();
    should_be_series(&else_series);
    should_be_fi_symbol();
```



```
    enter_4(IFMARK,cond_series,then_series,else_series,v);
    return SUCCEED;
  };
 return FAIL;
}
```

The semantic actions in this example are:

```
void enter_pair(int u,int v,int *w)
void enter_4(int mark,int field1,int field2,
             int field3,int *v)
```

which can be seen as multiple push operations on a stack in which the parse tree is built. Actually, this stack has no `pop`-operation, during its lifetime it only grows, and at the end it is discarded (a **quasi stack**). Notice that these actions build the tree *inside out*, in the sense that each node is entered only after all its sons have been entered. This is **inside-out property** which we will exploit later.

```
static int tree[MAXTREESIZE];
static int tree_top=0;

void enter_pair(int u,int v,int *w)
{
  assert(tree_top<=(MAXTREESIZE-2));
  *w = tree_top;
  tree[tree_top++] = u;
  tree[tree_top++] = v;
}

void enter_4(int mark,int field1,int field2,int field3,int *v)
{
  assert(tree_top<=(MAXTREESIZE-4));
  *v = tree_top;
  tree[tree_top++] = mark;
  tree[tree_top++] = field1;
  tree[tree_top++] = field2;
  tree[tree_top++] = field3;
}
```

A variant of this strategy may first reserve space for the node and fill it in later, after it sons have been recognized, where the semantic action `tie` "ties" the loose ends together by filling slots in the node pointed to by `v`.

6.3 List structures

A drawback of trees with explicit pointers is the large number of pointers needed. We may recall that in a regular t-adic tree only $1/t$ of the pointers are not nil. Of course the structures used here are not regular, and will be chosen such that few nil fields occur, but it is worthwile to look for representations with a small t: linear lists and binary trees.

As an example, the unit

```
X := IF a > b THEN a*b+c ELSE a*b-c FI
```

will lead to a storage structure like

```
0: [ :=, x, 1 ]
1: [ IF, 2, 3, 4 ]
2: [ 5, NIL ]
5: [ >, a, b ]
3: [ 6, NIL ]
6: [ +, 7, c ]
7: [ *, a, b ]
4: [ 8, NIL ]
8: [ -, 9, c ]
9: [ *, a, b ]
```

Notice that of the 28 fields 12 are pointers, of which 3 are nil. The advantage of having so many explicit pointers is that structural transformations on the parse trees, such as the insertion of explicit coercion operators or the restructuring for optimization purposes, can be performed easily. The disadvantage is the excessive memory requirements for the parse tree. In case we have no intention to apply structural transformations to the parse tree, we may as well try to omit all explicit references, and look at implicit tree representations.

6.4 Implicit Trees

An **implicit tree** is a tree represented without references, another relation (e.g. the ordering of elements in an array) is used instead.

6.4.1 Postfix form

In parsing a construct having various alternatives, we do not in general know the type of the node to be entered into the parse tree until all component subtrees have been recognized and therefore (since constructing the parse tree is a side-effect of recognition) have been entered. There is therefore a natural relationship between a **postfix traversal** of the parse tree (in which each subtree of a node is visited before its root is visited) and the construction of the parse tree. The semantic action for entering a node is then called after all operands have been translated.

```
a + b               ->     a b +
3 * a + 4 * b       ->     3 a * 4 b * +
```

In the abstract syntax tree, an expression is either a single primary or a sequence of n expressions (the operands) followed by an operator of arity n (where n is usually 1 or 2). This postfix form can been seen as an implicit tree, i.e. a tree without explicit references (as before) or brackets (as necessary in the infix form). The postfix form is used by some computers even for their machine code, and can be seen as the instruction code for a stack machine.

Recursive descent recognizers can easily be extended with postfix parse tree generation as a side effect:

```
ACTION expression :
  term,
    (is adding operator (OP), expression, produce (OP);
     +).

ACTION term:
  is denotation (V), produce (V) ;
  is variable (V), produce (V) ;
  is open, expression, is close ;
  error("Incorrect expression").
```


The semantic action `produce (V)` is a push-operation on the stack in which the parse tree is built. The tree can be re-parsed easily (but only in the reverse order) starting from the top of the stack, which acts as root of the tree.

The postfix form is not uniquely determined for chain expressions: should `a + b + c` be rendered as `a b c + +` or as `a b + c +`? The parser given above yields the first representation. The second has the slight advantage of requiring somewhat less stack space (both for recognition and re-parsing). It can be achieved by replacing the recursion by iteration.

A problem with reparsing the postfix form is that the absence of explicit pointers makes it hard to find the end of a construct, in order e.g. to skip over a subtree, as in finding the condition in `cond then else IF`. For lists without a fixed length, a postfix endmarker indicating the length will have to be introduced, e.g. for representing a series

```
unit$_1$   unit$_2$   ...   unit$_n$  n
```

In such cases the temptation is large to stray from the pure postfix schema.

Postfix form programs can be executed or interpreted bottom-up by a push-down automaton, provided the semantics of operands in a formula does not depend on the operation (**compositionality** of semantics).

6.4.2 Prefix form

In the **prefix form** ("Polish form"), an operator of arity n precedes its n operands. Control structures can similarly be seen as operators, each with some arity. Prefix form can again be seen as an implicit prefix tree and needs no brackets.

```
a + b          -> + a b
3 * a + 4 * b  -> + * 3 a * 4 b
```

Prefix trees are easy to reparse and to interpret top-down, since the node markers at convenient places should preclude all ambiguity. The parse tree is generated essentially as in the first section of this chapter, but without storing pointers.

Producing the prefix form of an infix expression is complicated by the presence of priorities and bracketing. As an example, consider the following traditional grammar for expressions. XXX

```
E: T;       $<$ T $>$
   E + T.  $<$ + E T $>$

T: F;       $<$ F $>$
   T * F   $<$ * T F $>$

F: var;     $<$ Vn $>$
   den.     $<$ Dn $>$
```

We have marked each alternative with its intended prefix translation. Now consider the translation of the two expressions

```
v * v + d   ->   + * v v d
v * v       ->   * v v
```

Depending on the right context, either one or two operators should precede the first variable. In a system with 10 priorities (the example has only two), up to 10 operators may precede the first translated primary. A similar complication (but this time with unbounded depth) occurs with the unbounded nesting bracketed expressions. Therefore prefix form generation is not in general a linear one-pass process.

To close this section with a positive note, both the postfix and prefix representations are quite dense, as is shown by our running example

```
:= x IF > a b NIL + * a b c NIL - * a b c NIL
```

in which only 3 of 19 cells are (unnecessary) overhead.

6.4.3 Mixing prefix and postfix form

Prefix form is more congenial than postfix form; for control structures prefix form is easy enough to generate, but for expressions it is not simple to obtain. A compromise therefore is to produce an implicit postfix tree during one scan and then to read it *backwards* during subsequent scans. The resulting "quasi prefix" order is not the same as prefix order because the operands (and e.g. the units of a series) are in reverse order. With a judicious use of recursion, the artifacts of order can be eliminated. For two-pass interpreters and translation this compromise may well be completely adequate.

6.5 Transducers to abstract syntax

In this section we will pay attention to the syntax-directed nature of parse tree generation and the restraints it imposes on the abstract syntax.

6.5.1 Abstract and concrete syntax

In the beginning of this chapter we pointed out that the **concrete syntax** of a language may not be the best basis for the internal storage of the parse tree, due to its redundant nature: It is intended to describe a linear input and therefore contains al kinds of bracket structures that can be more concisely represented by a prefix operator of suitable arity. Furthermore, the concrete syntax is a description serving other purposes than optimizing the internal representation; for didactic reasons and due to limitations (especially of CF grammar) it may contain redundant productions and chain rules that, if represented in the parse tree, would clutter up storage.

It is therefore important to devise a suitable **abstract syntax**, which not only eliminates such redundancies but also solves any (local) ambiguities in the concrete syntax by suitable prefix markers. The resulting abstract syntax tree is intended to be concise and easy to reparse.

On the other hand, the further the abstract syntax is from the concrete syntax, the harder it is to prove equivalence and correctness of interpretation or translation. The abstract syntax has to be designed with care: it is the first important **intermediate language** in a compiler and the starting point for many phases concerning identification, context condition and code generation. In particular it may be necessary to leave "holes" in the abstract syntax tree for later phases (e.g. for the type of some construct).

As an example we will consider a possible abstract syntax for part of Lalgol68. In the process we will introduce a notation for transducers that is quite useful in describing simple translation processes.

6.5.2 Transducers

A transducer is a mapping from strings in one language to string in another language. The simplest form of transducers can be obtained by distinguishing two kinds of terminal

symbols in a CF grammar, the input- and the output-alfabet (a **simultaneous grammar**).

Any sentence in such a simultaneous CF grammar can be interpreted as a pair of sentences, one in the input- language and one in the output-language. These sentences may be seen as the input and output of the transducer described by that CF grammar. As an example, the highly ambiguous grammar with $V_{in} = \{a\}$ and $V_{out} = \{x, |\}$

```
S:  a, x;
    a, x, S;
    S, $|$, S.
```

describes all possible nonempty partitionings of n a's. As an example, aaaa transduces to

```
xxxx
x|xxx     x|x|xx     x|xx|x     x|x|x|x
xx|xx     xx|x|x
xxx|x
```

In general we are interested in **deterministic transducers**, described by an unambiguous grammar.

The simultaneous grammar can express only extremely simple transductions, involving only a change in alphabet, but cannot express e.g. a change in the order of constituents. A slightly more powerful schema, syntax directed transduction, is based on the idea of simultaneous rewriting in two different grammars.

6.5.3 Transducer notation

We introduce here an informal notation for Context-Free **syntax-directed transducers** (**SDT**'s) as an extension to a CF grammar, which is highly suitable for description and documentation purposes. This notation shows some resemblance to Lcdl3, which was inspired by it.

- In principle we indicate, next to each alternative, its translation as a tuple. In the translation all the nonterminals of the alternative may be used (at most once but in any order), as well as any terminal symbol of the output language. The appearance of some nonterminal on the right denotes the translation of the same nonterminal on the left.
- Input terminal symbols have no translation, but output terminal symbols may be inserted at will.
- If the translation of an alternative consists merely of the translations of its members, in that order, we omit it (to save writing).
- If the translation of all alternatives of a rule is the same, we indicate it once and for all next to the lefthand side of the rule.

Notice that the appearance of some tuple as a translation (e.g. `< WHILE, condition, do part >`) says nothing about the way such a tuple is represented internally (with explicit references, in prefix or in postfix form); this is an issue that the notation abstracts from.

6.5.4 Transducer example

Let us now consider a transducer from concrete Lalgol68 to an abstract syntax. We will start with the syntax of `program`.

```
program:
   enclosed clause. $<$ PROG, enclosed clause $>$
```

The translation of a program is the marker `PROG` followed by the translation of the enclosed clause of which it consists.

```
enclosed clause:
    conditional clause;
    collateral clause;
    closed clause;
    case clause.
```

Since we do not give any translation explicity, by default the translation of an `enclosed clause` will be that of its direct production. As a consequence, the notion of `enclosed clause` will not occur explicity in the abstract syntax!

```
conditional clause:
   if symbol, choice, fi symbol.
```

The translation will be that of the `choice`.

```
choice:  $<$IF, condition, then part, else part$>$
    condition, then part, else part.
```

This introduces a tuple with marker `IF`.

```
condition: serial clause.
then part: then symbol, serial clause.
```

Notice that the `then symbol` will not appear in the abstract syntax.

```
else part:
    else symbol, serial clause;
    elif symbol, choice;
    .                        NIL
```

For the first two alternatives, the translation is that of their (only) constituent nonterminal. The third alternative inserts the marker NIL into the output. In order to keep the transducer simple, we have not introduced the Lalgol68 shorthand notation `(...|...|...)` for

IF ...THEN...ELSE...FI

Now for a tougher problem.

```
collateral clause:  $<$ COLL, unit list $>$
    begin symbol, unit list, end symbol;
    open symbol, unit list, close symbol.
```

Since both alternatives generate the same translation, we have indicated it only once, at the heading of the rule.

```
unit list:
    unit, comma symbol, unit list;  $<$ unit, unit list $>$
    unit.  $<$ unit, NIL $>$
```

The abstract syntax here amounts to a linear list of units.

```
closed clause:  $<$ CLOS, serial clause $>$
    begin symbol, serial clause, end symbol;
    open symbol, serial clause, close symbol.
```

Notice that we now have a serious prefix sharing problem with `collateral clause` syntax. For the transducer it is irrelevant how this problem is solved, since it describes a

relationship between strings and trees, not a translation process.

```
serial clause:
    declaration, semicolon symbol,
       serial clause;   $<$ declaration, serial clause $>$
    unit, semicolon symbol,
       serial clause;   $<$ unit, serial clause $>$
    unit.  $<$ unit, NIL $>$
```

The last two alternatives share `unit` as prefix. Just like in the realization of `unit list` some form of left-factorization will be necessary. Notice that the trees generated for a `serial clause` are indistinguishable from those generated for a `unit list` since the distinguishing markers `(` `;` and `)` have been elided.

```
unit:
   enclosed clause;
   expression.
```

The transducers for `expression` and `declaration` we shall omit; the reader may tackle them as an exercise. We hope to have shown that this form of transducers is indeed a suitable high level notation for simple translations.

6.5.5 The resulting abstract syntax

The resulting abstract syntax can be deduced from the transducer by considering only the nonterminals and the translations parts of the rules. This leads to:

```
program: $<$ PROG, unit list $>$.
```

Here we make use of the fact noted above that the translation of a `unit list` is indistinguishable from that of a `serial clause`; in the abstract syntax, we will consider the translation of a `serial clause` to be a `unit list`. We therefore generalize the notion of `unit` to include `declaration`s.

```
unit:
    enclosed unit;
    declaration;
    expression.

unit list:
   $<$ unit, unit list $>$;
   $<$ unit, NIL $>$.
```

The abstract syntax for `expression`, which we do not consider here, should be such that the translation of an `enclosed clause` or `declaration` can immediately be distinguished from that of an `expression` (e.g. by a prefix marker).

```
enclosed unit:
    $<$ IF, condition, then part, else part $>$;
    $<$ COLL, unit list $>$;
    $<$ CLOS, unit list $>$.

condition: unit list.
then part: unit list.
else part: unit list.
```

As can be seen, the resulting abstract syntax is much simpler than the original concrete syntax. Furthermore it is in prefix form and LL(1), which should make reparsing easy.

6.6 Exercises

1. Design an algorithm for the in-situ conversion of a postfix tree into a prefix tree.
2. Design suitable transducers for `expression` and `declaration`.

Chapter 7

IDENTIFICATION

Each occurrence of a name in a program is either *defining* or *applying*, e.g. in

```
BEGIN INT i; ...   i := i + 1; ...   END
```

the first occurrence of `i` is defining, the others are applying.

The **defining occurence** (in a declaration) of an object specifies its name and type. All other occurrences (outside comments and test-denotations) are **applied occurrences**. For every **application** of a name the compiler has to determine which **definition** this application belongs to, according to the **visibility** rules of the language. The compiler thus obtains information about the identity and properties of the object. This process is called **identification.**

7.1 Block-structure

Languages with **block structure** allow a recursive structure of **blocks** (procedure bodies, etc.) delimiting the visibilty. Some terminology:

- By the **static blocknumber** of a block we mean the nesting level according to the block structure (the number of enclosing blocks).
- By the **dynamic blocknumber** we mean an unique number associated with this block.

A **canonical blocknumbering**

```
0 begin 1
    1 begin 2
    2 end 1
    1 begin 3
        3 begin 4
        4 end 3
    3 end 1
    1 begin 5
    5 end 1
1 end 0
```

can be established as follows:

```
MODULE blocknumbering = FATHER{} and CURRBN and NEXTBN

DEFINES BN,
        TEST has father(>BN1,BN2>),
        FUNCTION get blocknr(BN>),
        ACTION enter block,
        ACTION leave block.

BN, CURRBN, NEXTBN = INT.

PRELUDE in outermost block:
  [0->CURRBN],
    [1->NEXTBN].

ACTION enter block:
  [CURRBN->FATHER{NEXTBN}],
    [NEXTBN->CURRBN],
    [NEXTBN+1->NEXTBN].

ACTION leave block:
  [FATHER{CURRBN}->CURRBN].

TEST has father(>BN1,BN2>):
  [FATHER{BN1}->BN2],
    greater(BN2,0).

FUNCTION get blocknr(BN>):
  [CURRBN->BN].
```


The resulting **father structure** for the above example is shown in figure 7.1.

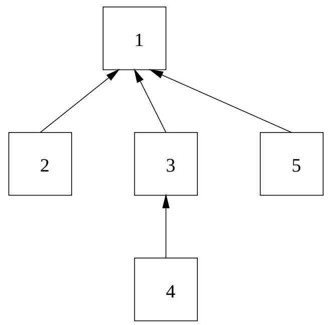


Figure 7.1: Father-structure

7.2 Identification in block-structured languages

We are concerned with the storage and retrieval of context information associated with names occurring in various blocks. To that end, we have to maintain a changeable mapping

$$identification : name * environment \mapsto type$$

where the *name* may e.g be an identifier, the *environment* a dynamic blocknumber. We realize it by the following interface:

- At defining occurrence

```
ACTION define type (>NAME, >BN, >TYPE):
```

- At applied occurrence

```
PRED obtain type (>NAME, >BN, TYPE>):
```

7.3 One-pass identification

7.3.1 Simple scheme

A simple **one-pass identification** scheme can be given for the following situation:

- each name has precisely one definition
- each name has one or more applications textually following the definition.

This simple scheme is in general not useable:

- applications of names may precede their definition (examples: labels and gotos, recursion).
- different environments may be valid within one same program (block structured languages, local names in Lcdl3 modules or ELAN packets).

Such complications make it impossible to use a one-pass translation scheme since the information necessary for correct identification is known too late in the translation process. There are languages which introduce **FORWARD declarations** and declarations of labels, to allow one-pass translation.

7.3.2 improvement

Restriction: definitions must precede applications. Each identifier has at most one declaration per block.

Associate a stack of attributes with each name. Push at every definition, pop at end of corresponding block

```
BEGIN
   def x
   BEGIN
      appl x   illegal
      def x    push old type of x
      appl x
   END         pop old type of x
   appl x
END
```

How to catch illegal applications? A partial solution can be found by remembering the bn of the last application.

7.4 Two-pass identification

For a full solution it is better to introduce a **two-pass model**: treat definitions in the **context-free analysis** pass, applications in a **context-sensitive analysis** pass. Two strategies can be distinguished:

- to associate a namelist with each block
- to associate a list of defining occurrences with every idf.

7.4.1 A namelist with each block

Maintaining a changeable mapping

$$namelist : bn \mapsto (name, type)^*$$

```
MODULE identification1 = table ENV{}.
DEFINES TYPE,
        NAME,
        ACTION define type(>NAME,>BN,>TYPE),
        TEST obtain type(>NAME,>BN,TYPE>).

USES blocknumbering.

DEF  :: NAME TYPE.
ENV, NEST :: nil; DEF NEST.

ACTION define type(>NAME,>BN,>TYPE):
  in(NAME,ENV{BN},TYPE1),
    error("Duplicate definition of " NAME );
  [NAME TYPE ENV{BN}->ENV{BN}].

TEST in(>NAME,>NEST,TYPE>):
  [NEST ->NAME1 TYPE NEST1],
    ([NAME=NAME1];
     in(NAME,NEST1,TYPE)).

TEST obtain type(>NAME,>BN,TYPE>).
  in(NAME,ENV{BN},TYPE);
  has father(BN,BN1),
    obtain type(NAME,BN1,TYPE).
```

7.4.2 A list of definitions with every name

Maintaining a changeable mapping

$$namelist : name \mapsto (bn, type)^*$$

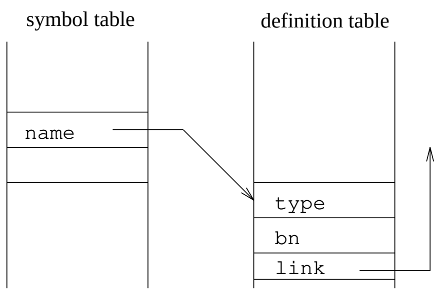


Figure 7.2: List of definitions for name

```
MODULE identification2 = symbol table SYMBOLS
     containing NR_OF_SYMBOLS with PROPS{}.

DEFINES
        TYPE, NAME,
        ACTION define type(>NAME,>BN,>TYPE),
        TEST obtain type(>NAME,>BN,TYPE>).

DLIST   :: nil; PROP DLIST.
PROP    :: NAME NR.
NLIST   :: nil; DEF NLIST.
DEF     :: BN TYPE.
NR      =  INT.

PROPS   =  NLIST.
SYMBOLS =  DLIST.

ACTION define type(>NAME,>BN,>TYPE):
  get props(NAME,NR),
    (in(PROPS{NR},BN,TYPE1),
       error("Duplicate definition of " NAME );
     add(PROPS{NR},BN,TYPE));
  new definition(NAME,NR),
    add(PROPS{NR},BN,TYPE).

TEST in(>NLIST,>BN,TYPE>):
  [NLIST->BN1 TYPE NLIST1],
    ([BN1=BN];
     in(NLIST1,BN,TYPE)).

FUNCTION add(>NLIST>,>BN,>TYPE):
  [NLIST->BN1 TYPE1 NLIST1],
    greater(BN1,BN),
    add(NLIST1,BN,TYPE),
    [BN1 TYPE1 NLIST1->NLIST];
  [BN TYPE NLIST->NLIST].

ACTION new definition(>NAME,NR>):
  [NR_OF_SYMBOLS->NR],
    [NR_OF_SYMBOLS+1->NR_OF_SYMBOLS],
    [NAME NR SYMBOLS->SYMBOLS],
    [nil->PROPS{NR}].

TEST in(>DLIST,>NAME,NR>):
  [DLIST->NAME1 NR DLIST1],
    ([NAME1=NAME];
     in(DLIST1,NAME,NR)).

TEST get props(>NAME,NR>):
  in(SYMBOLS,NAME,NR).

TEST obtain type(>NAME,>BN,TYPE>):
  get props(NAME,NR),
    obtain type(PROPS{NR},BN,TYPE).
```



```
TEST obtain type(>NLIST,>BN,TYPE>):
  [NLIST->BN1 TYPE NLIST1],
    ([BN1=BN];
     less(BN1,BN),
       (has father(BN,BN2),
          obtain type(NLIST,BN2,TYPE);
        -);
     in(NLIST1,BN,TYPE)).
```

7.5 Representing types

Types can be represented in a number of ways, e.g. as declarers embedded in the parse tree of the program, or in a separate type table or declarer table.

7.5.1 Types

The type system in ALGOL68 is relatively rich:

- rich set of type constructors
- user-defined types
- recursive types.

Therefore we will take it as example.
The **type table** is a datastructure that yields a key for each type, such that

- $type(x) = type(y) \Leftarrow key(x) = key(y)$
- key allows printing, etc
- for every applicable **coercion** (see later) there is a function like

$$deref : key \to key, ...$$

  yielding the resulting type.

There is a problem: equivalence of types involving **abstract typess** needs identification, but identification comes one pass later. A heuristic solution:

> if two declarers are the same sequence of symbols and they do not contain abstract types, then they are certainly equivalent.

This catches already most equivalences. But we will do better.

7.5.1.1 Metasyntax of types in A68

We quote the metasyntax of types from the Lalgol68 report, (replacing the term `MODE` by `TYPE`).

```
TYPE :: PRIMITIVE; reference to TYPE;
   LONGS INTREAL; PROCEDURE;
   structured with FIELDS;
   ROWS of TYPE; union of TYPES type.

PRIMITIVE :: INTREAL; bool; char.
INTREAL ::  int; real.
LONGS :: long LONGS; long.
```

```
PROCEDURE :: procedure PARAMETY MOID.
PARAMETY :: with PARAMETERS; EMPTY.
EMPTY ::.
PARAMETERS :: PARAMETER and PARAMETERS; PARAMETER.
PARAMETER :: TYPE parameter.

ROWS :: row ROWS; row.

FIELDS :: FIELD and FIELDS; FIELD.
FIELD :: TYPE field TAG.

TAG :: letter LETTER; TAG letter LETTER; TAG digit DIGIT.
LETTER :: a; b; ...; z.
DIGIT :: 0; 1; 2; 3; 4; 5; 6; 7; 8; 9.

TYPES :: TYPE and TYPES; TYPE.

MOID ::  TYPE; void.
```

7.5.1.2 A representation for types

We enter both types and their fragmentary constituents into the type table in the form of (head, tail) pairs, by means of

```
ACTION enter (>DEC, KEY>):
```

For every type we obtain a unique key.

type fragment	is stored as
`integer`	(INT, 0)
`real`	(REAL, 0)
`boolean`	(BOOL, 0)
`character`	(CHAR, 0)
`void`	(VOID, 0)
abstract type ABC	(ABC, dynamic bn)
`reference to TYPE`	(REF, TYPE)
`long TYPE`	(LONG, TYPE)
`procedure PARAMETY MOID`	(PROC, (PARAMETY, MOID))
`PARAMETER and PARAMETERS`	(PARAMETER, PARAMETERS)
`PARAMETER`	(PARAMETER, NIL)
`EMPTY`	NIL
`structured with FIELDS`	(STRUCT, FIELDS)
`FIELD and FIELDS`	(FIELD, FIELDS)
`FIELD`	(FIELD, NIL)
`TYPE field TAG`	(TAG, TYPE)
`row of TYPE`	(ROW, TYPE)
`union of TYPES` type	(UNION, TYPES)
`TYPE and TYPES`	(TYPE, TYPES)
`TYPE`	(TYPE, NIL)

Notice that, for abstract types, we have to take into account also the dynamic blocknumber of the occurrence of the declarer, because it enters into the identification. Concrete types are treated as if they occur in block zero, the standard library.
The representation described here has the property

$$\textit{equal representations} \Rightarrow \textit{equal keys} \Rightarrow \textit{equal types}$$

but not

$$\textit{equal keys} \Leftarrow \textit{equal types}$$

Determining type equivalence is to be done by a separate phase.

7.5.2 Representation in Łcdl3

Inspired by the previous representation, we can model the Algol68 type system (including the lists of types and fileds) in CDL3 as follows:

```
MODULE type handling = DECS{} with NR_OF_DECS.
DEFINES DEC,
        KEY,
        ACTION enter(>DEC,KEY>),
        FUNCTION retrieve(>KEY,DEC>).
```

A type is represented by an (abstracted) declarer, which has the following metasyntax:

```
DEC:: primitive PRIM;
      indicant NAME BN;
      reference to DEC;
      long DEC;
      procedure with PARS RES;
          and DEC DEC;nil;
      structered with FIELDS;
          and FIELD FLIELDS;
      row INT DEC;
      union DECS.

PRIM: int; real; bool; char; void.
```

Notice that we have here included `VOID` among the primitive types. The following synonyms were introduced to keep the previous metasyntax readable:

```
PARS,RES,FIELDS,DECS = DEC.
BN = INT

KEY,L,U :: INT.
NR_OF_DECS = KEY.
```

We define a (rather simplistic) type table:

```
ACTION enter(>DEC,KEY>):
  can find(1,NR_OF_DECS,DEC,KEY);
  [NR_OF_DECS->KEY],
    [NR_OF_DECS+1->NR_OF_DECS],
    [DEC->DECS{KEY}].

TEST can find(>L,>U,>DEC,KEY>):
  less(U,L),-;
  [DECS{L}=DEC],[L->key];
  can find(L+1,U,DEC,KEY).
```

7.5.3 Declarers

The following syntax is an excerpt from the unrevised ALGOL68 Report:

```
TYPE declarer : TYPE declarator; applied TYPE indicant.
```

```
PRIMITIVE declarator : PRIMITIVE symbol.
```

Example: `INT`

```
long INTREAL declarator : long symbol, INTREAL declarator.

long LONGS INTREAL declarator :
       long symbol, LONGS INTREAL declarator
```

Example: `LONG LONG INT`

```
reference to TYPE declarator:
      reference to symbol, TYPE declarer.
```

Example: `REF CHAR`

```
structured with FIELDS declarator :
     structure symbol, open symbol, FIELDS part,
     close symbol.

TYPE field TAG part : TYPE declarer, TAG.

TYPE field TAG and FIELDS part :
    TYPE declarer, TAG, comma symbol,
        FIELDS part.
```

Example: `STRUCT (REAL re, REAL im)`

- the syntactic shortening conventions ("extensions") have not yet been included here
- the grammar is a transducer between meta notions and terminal productions
- incomplete, but representative for the rest.

7.5.4 Transducing types to representations

The following Lcdl3 program attempts to follow as closely as possible the two-level grammar from the previous section.

```
MODULE declarer reading.

DEFINES
   TYPE,
   PRED is declarer (TYPE>).

USES type handling.

M, PAR, PARS, FIELD, FIELDS, TYPE = DEC.

PRED is declarer (TYPE>):
   is primitive declarator (TYPE>);
   is type indication (TYPE>);
   is long declarator (TYPE>);
   is reference declarator (TYPE>);
   is procedure declarator (TYPE>);
   is structure declarator (TYPE>);
   is row declarator (TYPE>);
   is union declarator (TYPE>).
```



```
PRED is primitive declarator (TYPE>):
   is ("INT"), enter (primitive int, TYPE);
   is ("REAL"), enter (primitive real, TYPE);
   is ("BOOL"), enter (primitive bool, TYPE);
   is ("CHAR"), enter (primitive char, TYPE);
   is ("VOID"), enter (primitive void, TYPE).

PRED is type indication (TYPE>):
   is bold (NAME),
      get blocknumber (BN),
         enter (indicant NAME BN, TYPE).

PRED is long declarator (TYPE>):
   is ("LONG"),
      (is long declarator (TYPE),
         enter (long TYPE, TYPE);
       is intreal declarator (TYPE),
         enter (long TYPE, TYPE);
       syntax error ("Incorrect LONG declarer"),
         dummy (TYPE)).

FUNCTION dummy (nil>): .

# ----------------- references------#

PRED is reference declarator (TYPE>):
   is ("REF"),
      should be declarer (TYPE),
         enter (reference to TYPE, TYPE).

ACTION should be declarer (TYPE>):
   is declarer (TYPE);
   syntax error ("Declarer expected"),
      dummy (TYPE).

# ------------------------------- procedures--------#

PRED is procedure declarator (TYPE>):
   is ("PROC"),
      may be parameters pack (PARS),
         may be result (RES),
            enter (procedure with PARS RES, TYPE).

ACTION may be parameters pack (PARS>):
   is ("("),
      should be parameters (PARS),
         should be (")");
   dummy (PARS).

ACT should be parameters (PARS>):
   parameter (PAR),
      rest parameters (PARS1),
         enter (and PAR PARS1, PARS).

ACT parameter (PAR>):
```

```
   is declarer (PAR);
   syntax error ("Parameter expected"),
      dummy (PAR).

ACT rest parameters (PARS>):
   is (","),
      should be declarer (PAR),
         rest parameters (PARS1),
            enter (and PAR PARS1, PARS);
   dummy (PARS).

ACT may be result (TYPE>):
   is declarer (TYPE);
   [primitive void -> TYPE].

# ------------------------------- structures -------#

PRED is structure declarator (TYPE>):
   is ("STRUCT"),
      rest structure declarator (M),
         enter (structured with M, TYPE).

ACTION rest structure declarator (TYPE>):
   is ("("),
      field (M, FIELD),
         rest fields (M, FIELDS),
            enter (and FIELD FIELDS, TYPE);
   syntax error ("Incorrect STRUCT declarer"),
      dummy (TYPE).

ACTION field (TYPE>, FIELD>):
   is declarer (TYPE),
      must be fieldselector (TYPE, FIELD).

ACTION must be fieldselector (>TYPE, FIELD>):
   is identifier (NAME),
      enter (field TYPE NAME, FIELD);
   syntax error (Missing fieldselector"),
      dummy (FIELD).

ACTION rest fields (>M, TYPE>):
   is (","),
      (is identifier (NAME),
          enter (field M NAME, FIELD),
             rest fields (M, FIELDS),
                enter (and FIELD FIELDS, TYPE);
      field (M, FIELD),
         rest fields (M, FIELDS),
            enter (and FIELD FIELDS, TYPE));
   is (")"),
      dummy (TYPE);
   syntax error ("Incorrect STRUCT declarer"),
      dummy (TYPE).
```


We have constructed a transducer from the concrete syntax of Lalgol68 to an abstract syntax of types.

Chapter 8

GENERIC IDENTIFICATION

We shall call an entity **generic** if more than one definition for it is visible in an environment. In the opposite case, where only one definition is visible, we call it **unique**. In modern languages, operators and procedures may be generic. Some languages like Lada allow **overloading** only for the standard operators.
In this chapter we investigate the identification of generic operators. Expressions in a program contain applied occurrences of operators:

```
a + b * c
```

Each applied occurrence of an operator should identify one out of a collection of definitions for that operator. In some languages, operator identification is complicated by the fact that the priority of operators may be defined by the user. We furthermore describe the problems associated with coercions. Finally, we deal shortly with polymorphic identification.

8.1 Priority declarations

Standard operators usually have standard priorities. But what is the priority of user-defined operators ?

- Lalgol68
  Has **priority declarations**, associating a one-digit number with an operator, like

  ```
  PRIORITY SUB = 3
  ```

  Each operator declaration is therefore an applying occurrence of some priority declaration, according to the visibility rules.

  Because priority declarations may change completely the interpretation of an expression, a priority declaration makes all operator declarations for that name in surrounding blocks invisible.

  - Pro : works beautifully to model the standard priorities.
  - Con : so what? what does priority 3 mean?
  - Also some ambiguities arize:

    ```
    BEGIN SUB x ;...; MODE SUB = INT;  ...END
    BEGIN SUB x ;...; PRIORITY SUB = 7;  ...END
    ```

- Lelan
  All dyadic abstract operators have the same (low) priority, just above assignation.

  - pro : no natural priority in most cases
  - con : sometimes irritating

    ```
    string SUB i + 1
    string SUB i + char
    ```

  - classical problem:

    ```
      - 1 ** 2
     (- 1) ** 2
      - (1 ** 2)
    ```

- Lalgol n
  In a more complete system, such as that proposed in 1967 by Yoneda for LalgolN, a Japanese version of Lalgol, not only the priorities of operators but also their binding directions can be specified. Operators in Lalgol n

  - may be polyadic
  - have distinct left-and right-priorities
  - may be left- or right associative.

  Also, the coercion strenght (8.4.1) of operands can be specified.

In a rich enough language the whole syntactic structure can be modeled by operators with suitable properties, which may be modified by the programmer. This idea leads to **extensible languages** eg. Lel 1(Cheatham around 1970), an approach which has now been relinquished: semantic extensibility and powerful abstraction mechanisms are for more important than syntactic extensibility.

8.2 Operator declarations

An operator declaration binds a certain **operation** (a piece of code) to an **operator** (a name for an operation).
Identification of generic operators is like the identification of unique operators: an **inside out** search for an **applicable** definition. A definition for an operator is applicable to a given application if, besides the name, the number and types of parameters agree.
Global declarations may be **shielded** by local applicable definitions. In one same block only one definition must be applicable; there are some **context conditions** which insist that identification must not be ambiguous. How do we check those conditions?

- check at definitions?
- check at applications?

Example
The following two declarations together in one block would lead to ambiguous identification:

```
OP + = (REF INT a) VOID : ... ;

OP + = (INT a) VOID: ... ;
```

when applied to an actual `REF INT` parameter.
The following couple would also appear ambiguous:

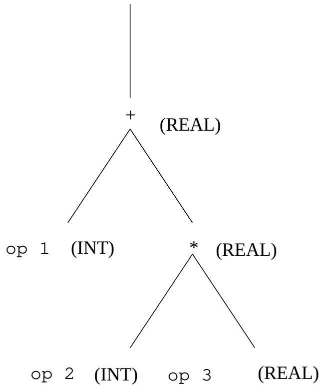
```
OP + = (REF INT a, INT b) VOID: ... ;

OP + = (REAL a, INT b) VOID: ... ;
```

but there exists no type of actual parameter that can be coerced to both `REF INT` and `REAL` (Lalgol68 does not allow widening on operand positions, sacrificing the symmetry with procedure parameters).

8.3 Identification and coercion

A **coercion** is an implicit type conversion, to be inserted at certain positions in the program. It can be seen as an invisible conversion operator.
Identification of procedure calls and operator invocations in expressions has to identify also the necessary coercions. The results of the identification process including coercions can be represented in various ways:

- decorating the operator-tree with types (bottom up)



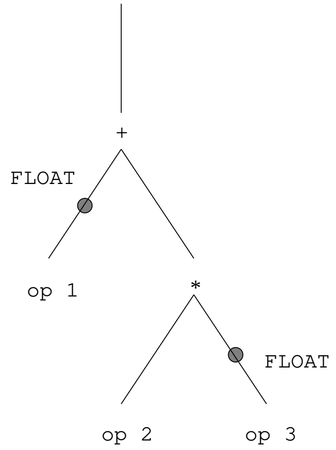
- patching the operator-tree by insertion of coercion nodes, or
- generating explicit coercion operators in the generated code.

8.3.1 Coercions in programming languages

Coercions had a long development history:

- Lfortran ii distinguished two different numeric types, `INTEGER` and `REAL`, allowing **widening** only at assignations. At other places, an explicit conversion had to be inserted.
- Lfortran iv allowed also "mixed-mode" expressions
- Lalgol60 had two different numeric types with widening in many cases, where for some operations the type of the result depended on the value of the operands (**dynamic typing**)

```
2  2     integer
2  -2    real
```

- In Lpl/1 (nearly) everything was coercable to everything

```
FIBONACCI ('     12')
```

- Lalgol68 brought system into this madness:
  - completely static typing
  - many different coercions, governed by "strength" of position

8.4 The Lalgol68 coercion machine

By the **a priori type** is meant the type of the coercend before coercion, the **a posteriori type** is the type obtained after coercion.
The basic ideas behind the coercions can be summarized as follows:

- 5 strengths of context
  - **strong position**
    a posteriori type known
  - **firm position**
    finite choice of a posteriori types
  - **weak position**
    distinction between variable and constant is to be maintained
  - **soft position**
    reference wanted
  - null
- 8 coercions
  - **dereferencing coercion**
    REF M $\Rightarrow$ M

- **widening coercion**
  INT ⇒ REAL
  CHAR ⇒ STRING
  INT ⇒ LONG INT
  REAL ⇒ COMPL
  In general: embedding of a type in a supertype; the inverse is called **narrowing**.
- **rowing coercion**
  M ⇒ [1:1] M
- **hipping coercion**
  VOID ⇒ M (for SKIP and JUMP)
- **voiding coercion**
  M ⇒ VOID
- **proceduring coercion**
  M ⇒ PROC M (not in revised language, see 8.4.6)
- **deproceduring coercion**
  PROC M ⇒ M
- **uniting coercion**
  M ⇒ UNION (..., M, ...)

- Only a **coercend** can be coerced: an `assignation`, `conformity relation`, `identity relation`, `cast`, `formula`, `generator`, `selection`, `slice`, `call`, `denotation` or `identifier`. Larger constructs are NOT coercends: `unit`, `expression`, `choice`, ...

8.4.1 Strength of context

The **strength** of a position indicates to what degree the type after coercion (the **a posteriori type**) is known, and determines the applicable subset of coercions.

8.4.1.1 Strong context

In a **strong context**, the a posteriori type is known, dictated entirely by the context. All coercions are applicable.

- units of a paragraph are void, except possibly the last
- right hand side of an assignation

```
var := expr
```

- rhs of an identity declaration

```
AMODE x = expr
```

- actual parameter of a procedure call
- in a cast

```
AMODE (expr)
```


8.4.1.2 Firm context

In a **firm context**, a (finite, small) choice between types is possible. Only proceduring (see 8.4.6), deproceduring, dereferencing, uniting are applicable.

- operands of (generic) operators
- contexts related to union-types.

8.4.1.3 Weak context

aA **weak context** occurs in positions where variables and constants must remain distinct. Therfore dereferencing may not take off the last reference.

```
a[i]   think of   p(x)[i]
field OF x   field OF p(x)
```

8.4.1.4 Soft context

A **soft context** occurs where a reference is wanted (e.g., in balancing reference levels in an `identity relator`).

```
REF REF REF REAL XXX=...;
REF REF REAL YY=...;
xxx :=: yy
```

8.4.1.5 Null context

No coercion.

8.4.2 Gradation of coercions

The coercions are grouped in levels of diminishing severity. The Lalgol68 report [ALG68] defines

```
ADAPTED :: ADJUSTED;
      widened; rowed; hipped; voided.
ADJUSTED :: FITTED;
      procedured; united.
FITTED :: dereferenced; deprocedured.
```

8.4.3 Gradation of strengths

```
SORT :: strong; FEAT.
SORTETY :: SORT; EMPTY.
EMPTY :: .
FEAT :: firm; weak; soft.
```

8.4.4 What gets coerced?

Not every construct can be subjected to coercion, but only a limited class of **coercends**. The metasyntax states

```
COERCEND :: MOID FORM.
FORM :: confrontation;
   ADIC formula;
   cohesion;
```

```
   base.
ADIC :: PRIORITY; monadic.
PRIORITY :: priority NUMBER.
NUMBER ::
   one;
   one plus one;
   .
   .
   .
   one plus one plus one plus one plus one plus
   one plus one plus one plus one plus one.
```

The syntax states

```
MODE confrontation:
   MODE assignation; MODE conformity relation;
   MODE identity relation; MODE cast.

MODE cohesion:
   MODE generator; MODE selection.

MODE base:
   MODE type identifier; MODE denotation;
   MODE slice; MODE call.

strong COERCEND:
      COERCEND;
      strongly ADAPTED to COERCEND.

firm COERCEND :
      COERCEND;
      firmly ADJUSTED to COERCEND.

weak COERCEND:
      COERCEND;
      weakly FITTED to COERCEND.

soft COERCEND: COERCEND;
      softly deprocedured to COERCEND.
```

In order to demonstrate the style in which the coercions are described syntactically, we show the syntax for dereferenced and procedured coercends.

8.4.5 Dereferenced coercends

```
STIRM :: strong; firm.

STIRMLY dereferenced to MODE FORM:
         reference to MODE FORM;
         STIRMLY FITTED to reference to MODE FORM.

weakly dereferenced to reference to MODE FORM:
      reference to reference to MODE FORM;
```



```
      weakly FITTED to reference to reference to MODE FORM.
```

Deproceduring is described similarly.

8.4.6 Procedured coercends

The proceduring coercion (dropped in the Revised Report [ALG68]) is the opposite of deproceduring, so one may wonder how it is described.

```
STIRMLY procedured to procedure MOID FORM:
   MOID FORM;
   STIRMLY dereferenced to MOID FORM;
   STIRMLY procedured to MOID FORM;
   STIRMLY united to MOID FORM;
   STIRMLY widened to MOID FORM;
   STIRMLY rowed to MOID FORM.
```

Notice that in this rule deproceduring is not mentioned, thus preventing coercion cycles. Even upreferencing might be in the language without paradoxes!

```
PROC REAL p := 3.14
```

is in unrevised A68 shorthand for

```
PROC REAL p := REAL : 3.14
```

From the above description, the Lalgol68 coercion machine can be derived as a nondeterministic automaton.

8.5 Polymorphic identification

The term **polymorphy** covers various situation in which one name is given to more than one entity, so that it has “multiple forms”.

- We have already dealt with generic operators.
- In e.g. Lfortran and Lalgol60 the whole numbers may appear both as integer and as real denotations (they are ambiguous denotations). The same effect is obtained by a widening coercion in Lalgol68.
- In Lalgol68 and Lelan, the display serves as a denotation for both rows and structures – the ambiguity is resolved by (strong) context.
- In principle even objects might be polymorphic, the context serving to resolve ambiguities.

We shall be concerned especially with **parametric polymorphism** (unfortunately called genericity in Lada), which is characterised by:

- Types may be explicitly parameterized by types (e.g. `LIST (REAL)`)
- Operations may be implicitly parameterized by types.

In this case, one definition is applicable to an open class of applications (with different types); contrast this with **genericity**, where each type of application needs a different definition.
An example of parametric polymorphism is found in the assignation, which will conventionally work for each (abstract or concrete) type. In pidgin-Lalgol68

$\forall M$ `M OP := = (REF M dest, M source) REF M: ...;`
An other conventional polymorphic operations is the subscription, which actually has two forms:

1. a dyadic value taker: $\bullet[\bullet]$
2. a triadic updater: $\bullet[\bullet] := \bullet$

In languages with parametric polymorphism (such as Lada, Lelan and many functional languages) the user can define his own polymorphic types and operations.

8.5.1 Example: polymorphic sort

In Lelan, the polymorphic type declaration

```
TYPE LIST(M) = STRUCT (M hd, LIST(M) tl);
```

introduces a parameterized type (an **abstract type constructor**) `LIST` with a free **type parameter** `M`. The procedure

```
PROC(M) sort (LIST(M) VAR l):
```

can be called with an actual parameter of a suitable parameterized type, e.g.

```
LIST (REAL) VAR measurements; ...;
sort (measurements)
```

in which case the type parameter `M` is bound to `REAL`.

```
TYPE LIST (M) = STRUCT (M hd, LIST (M) tl);

PROC (M) make empty (LIST (M) VAR l):
  l := NIL
ENDPROC make empty;

BOOL PROC (M) is empty (LIST (M) CONST l):
  ISNIL l
ENDPROC is empty;

M OP (M) HEAD (LIST (M) CONST l):
  l.hd
ENDOP HEAD;

LIST (M) OP (M) TAIL (LIST (M) CONST l):
  l.tl
ENDOP TAIL;

LIST (M) PROC (M) list (M CONST a):
  LIST (M): [a, NIL]
ENDPROC list;

LIST (M) OP (M) + (M CONST m, LIST (M) CONST l):
  LIST (M): [m, l]
ENDOP +;

PROC (M) print (LIST (M) CONST l):
  IF NOT is empty (l)
  THEN
    print (HEAD l);
    print (" ");
    print (TAIL l)
  FI
ENDPROC print;

PROC (M) sort (LIST (M) VAR l):
  IF is empty (l)
  THEN
  ELSE
    sort (l.tl);
    adjust (l)
  FI
ENDPROC sort;

PROC (M) adjust (LIST (M) VAR l):
  {sorted(l), nonempty(l)}
  IF is empty (l.tl)
  THEN
  ELIF l.hd < l.tl.hd
  THEN
  ELSE
    swap (l.hd, l.tl.hd);
    adjust (l.tl)
  FI
ENDPROC adjust;

PROC (M) swap (M VAR l, r):
  M CONST h :: l;
  l := r;
  r := h
ENDPROC swap;
program:
  LIST (REAL) VAR l;
  make empty (l);
  UPTO 10
  REP ll := random + l
  ENDREP;
  print (l);
  line (2);
  sort (l);
  print (l).
```


8.5.2 Implementation by expansion

In Lada and Lc++, polymorphic packets have to be expanded and coded separately for each parameter type. Standard identification techniques are to be applied after this **expansion**. Expansion is a simple but costly solution, since it may lead to a combinatorial explosion of the code size.

8.5.3 Necessary operations

A key observation leading to a more inventitive implementation is the fact that `LIST` and `sort` in the previous example can be parameterized with any type that possesses certain

necessary operations, e.g. an output procedure `print` for `M` required by `print` for `LIST(M)` and an ordering operator `<` on `M` required by the body of `sort`.
Such necessary operations can be considered as implicit parameters to the sorting procedure, which can be made explicit by passing the ordering operator explicitly to `sort` and thence to `adjust`, as in

```
PROC (M) sort (LIST (M) VAR l, OP (M CONST a,b)<):
   sort l in situ in increasing order;
   IF is empty (l) THEN
   ELSE    sort (l.tail);
      adjust (l, <)
   FI
ENDPROC sort;
```

In fact, this is a reasonable way to implement polymorphic operations:

- The values of the parameter-type are represented by references (**indirection**) which have a standard well-known size, so that values of all types can be passed as parameters, irrespective of their particular sizes

- A vector of addresses is passed for the necessary operations, the compiler has to construct one such vector for each actual parameter type.

Of course the indirection will cost some efficiency, but the flexibility and memory economy are much greater than in the case of expansion.
A similar implementation technique is used for **virtual procedures** in Object-Oriented Programming languages.

8.5.4 Ordering of types

How then should the identification take place in the face of overloading and parametric polymorphism? Again it is an inside-out process, looking for the nearest applicable definition, but now more than one definition in an environment or block may be applicable. In that case the **most defined** definition has to be taken, according to the following partial ordering of definedness:

- a concrete type, a strong abstract type or any type composed out of those is fully defined

- a weak abstract type is as defined as its realisation

- a free type parameter is least defined

- a type constructed out of a free type is more defined than a free type (e.g. `LIST (M)` is more defined than `M`).

It is therefore convenient to order all operations with the same name in order of decreasing definedness. This **definedness ordering** is a partial order even for monadic operations, and even more so for polyadic operations, which have to be ordered according to the lexicographical extension of the definedness ordering. The following examples show this ordering for various parameter types, first for some monadic operator, and then for some dyadic operator. In each case the examples on the first line are fully defined and each subsequent line is less defined than the previous.

(INT) (REAL) (ROW 10 INT) (LIST (INT))
(LIST(LIST(M)))
(LIST(M))
(M)
(INT, INT) (REAL, REAL) (REAL, INT) (LIST(INT), INT)
(INT, M) (M, INT)
(M,M)

Modern programming languages increasingly make use of a **subtyping** relation, which may add fine-structure to this partial ordering.

Chapter 9

TYPE EQUIVALENCE

In this chapter we describe techniques to establish whether two given types, represented by a pair of declarers in a program, are equivalent (**type equivalence**), either by considering one pair at-a-time ("incrementally") or once-and-for-all after all declarers have been collected.
We will describe two algorithms. But first we address the question *what types are allowed*?

9.1 wellformedness

The **wellformedness conditions** are context conditions on (sets of) type-declarations, such as

```
MODE M = ...,
     P = ...,
     Q = ... ;
```

where the left hand sides are (names of) abstract types and the right hand sides declarers specifying their domain (also called the **representation** or the **realization** of the abstract type).
The wellformedness conditions described in the Lalgol68 report have the following aims:

1. Type declarations corresponding to infinite objects are forbidden, such as

   ```
   MODE FULL = STRUCT ([1:10] FULL s, INT i)
   ```

   but allowed are

   ```
   MODE FULL = STRUCT (REF[1:10] FULL s, INT i)
   MODE FULL = STRUCT ([1:10] REF FULL s, INT i)
   MODE FULL = REF STRUCT ([1:10] FULL s, INT i)
   ```

   because the `NIL` reference can stop the recursion.

2. Type declarations that make a program ambiguous are forbidden. An example:

   ```
   MODE T = REF T;
   T t1, t2;  t1:= t2
   ```

   How many dereferencings in this assignation? Or, even worse:


   ```
   MODE T = T;
   ```

3. Type declarations are restricted to keep type equivalence decideable.

A recursive type declaration for `A` is **wellformed** if any (direct or indirect) occurrences of `A` in its realization are **shielded**, i.e.:

- preceded by (at least one) `REF`
- parameter of a procedure
- result of a procedure with parameters
- member of a union
- element of a row.

Forbidden is, apart from the examples above, e.g.

```
MODE P = PROC P;
MODE L = STRUCT (INT x, L y)
```

Allowed are

```
MODE LIST = REF STRUCT (INT x, LIST y)
MODE LIST = STRUCT (INT x, REF LIST y)
```

as well as **higher-order types** such as

```
MODE P = PROC (INT) P
```

and even

```
MODE P = PROC (P) P
```

which few other type systems allow. (This type looks very exotic, but it is used e.g. in constructiong **continuations** – a subject which will be dealt with in VB2)
It may be argued that the wellformedness conditions on types are still too weak, as shown by the apparently ambiguous example

```
MODE T = STRUCT (REF T t); Tt;
    t:= (t)
```

(is `(t)` a closed clause or a structure display of one field)? This ambiguity is resolved by demanding (syntactically) that all displays must have more than one constituent unit (therefore the example is a closed clause). A simpler solution (as in Lelan) is to use different brackets for closed clause and display.

9.2 K-algorithm

It is not immediately obvious that type equivalence in a recursive type system is decideable. As an example, the types of

```
MODE PERSON = STRUCT (REF PERSON x,
                      INT y);
MODE PARSON = STRUCT (REF PARSON x,
                      INT y);
MODE PURSON = STRUCT (REF STRUCT (REF PURSON x, INT y) x,
                      INT y);
```

should be equivalent, because they are obtained form another by the (intuitively harmless) transformations of substitution and renaming.
It turns out to be helpful to exploit the following intuition: Recursive type declarations specify an infinite tree. Two infinite trees must be considered equal if there is no (finite) difference between them to be found (see fig. 9.1).

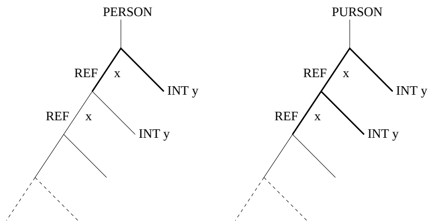


Figure 9.1: Comparing infinite trees

The K-algorithm [KOS69] is an algorithm to decide the pairwise equivalence of types, represented as declarers as described in table 7.5.1.2; the reader is invited to refresh his or her memory. Expressed in Lalgol68:

```
PROC equivalent = (DECLARER d1, d2) BOOL:
   IF same sequence of symbols (d1, d2)
   THEN TRUE
   ELIF is type indication (d1)
   THEN  IF has been postulated (d1, d2)
         THEN TRUE
         ELSE postulate (d1, d2);
            IF equivalent (defining declarer (d1), d2)
            THEN TRUE
            ELSE unpostulate (d1, d2);
                 FALSE
            FI
         FI
    ELIF is type indication (d2)
    THEN equivalent (d2, d1)
    ELSE same tree (d1, d2)
    FI;
```

where `postulate` and `unpostulate` administer an association list, and `same tree` is realized by a case distinction over the various forms of declarers, demanding equivalence of corresponding types and equality of corresponding names.

9.2.1 Examples

1. (Renaming)

```
MODE P = STRUCT (REF P x);
     Q = STRUCT (REF Q x);
```


postulates	hypotheses
	P ≡ Q?
P ≡ Q!	
	STRUCT (REF P x) ≡ Q?
	Q ≡ STRUCT (REF P x)?
Q ≡ STRUCT (REF P x)!	
	STRUCT (REF Q x) ≡ STRUCT (REF P x)?
	x = x
	REF Q ≡ REF P?
	Q ≡ P?
Q ≡ P!	
	STRUCT (REF Q x) ≡ P?
	P ≡ STRUCT (REF Q x)?
P ≡ STRUCT (REF Qx)!	
	STRUCT (REF P x) ≡ STRUCT (REF Q x)?
	x = x
	REF P ≡ REF Q?
	P ≡ Q?
	QED

(Unfolding)

```
MODE P = STRUCT(REF P x),
     M = STRUCT(REF STRUCT(REF M x) x)
```

postulates	hypotheses
	P ≡ M?
P ≡ M!	
	STRUCT(REF P x) ≡ M?
	M ≡ STRUCT(REF P x)?
M ≡ STRUCT(REF P x)!	
	STRUCT(REF STRUCT(REF M x)x) ≡ STRUCT(REF P x)?
	x = x
	REF STRUCT(REF M x) ≡ REF P?
	STRUCT(REF M x) ≡ P?
	P ≡ STRUCT(REF M x)?
P ≡ STRUCT(REF M x)!	
	STRUCT(REF P x) ≡ STRUCT(REF M x)?
	x = x
	REF P ≡ REF M?
	P ≡ M?
	QED

9.2.2 Paradox

Absolutely circular declarations like the following one for `M` are not wellformed. Admitting them would lead to strange consequences.

```
MODE   M = M, P = STRUCT(REF P x)
```

1. ⇒

postulates	hypotheses
	M ≡ P?
M ≡ P!	
	M ≡ P?
	QED

2. $\Leftarrow$

postulates	hypotheses
	P ≡ M?
P ≡ M!	
	STRUCT(REF P x) ≡ M?
	M ≡ STRUCT(REF P x)?
M ≡ STRUCT(REF P x)!	
	M ≡ STRUCT(REF P x)?
	QED

It turns out that `M` is equivalent to any type.

9.3 Deciding all equivalences

This algorithm can be employed to decide all equivalences. We show its operation on the example

```
MODE P = STRUCT (REF P x),
     Q = STRUCT (REF Q x)
```

We shall represent types by declarers, declarers by $< head, tail >$ pairs, stored from the inside out as described in chapter 6.2.3. Each pair is prefixed with a $< syn >$ field serving to store the postulates made and to link synonyms. The syn field will hold **downwards pointers** only, the field with the higher address pointing to the one with the lower, but not vice versa.

The algoritm is the following: Step 1: Enter all declarers inside out, storing only those pairs that differ.

	syn	head	tail
1:		P	$< bn >$
2:		REF	1
3:		2	x
P: 4:		STRUCT	3
5:		Q	$< bn >$
6:		REF	5
7:		6	x
Q: 8:		STRUCT	7

Figure 9.2: Two declarers entered

Step 2: Perform identification of type indicants, linking all applied type-indicants to their definition, by putting in the $< syn >$ field of the highest a pointer to the lowest. Step 3: try to establish all pairwise equivalences, proceeding from the lowest field upwards. The result should, in our example, look as in figure 9.4.

9.3.1 Realization in CDL3

We show a straightforward realization of this type equivalencing algorithm in Lcdl3, using the type table (indexed by keys) introduced in the previous chapter.

```
D, I, J:: KEY.
```


	syn	head	tail
1:		P	$< bn >$
2:		REF	1
3:		2	x
P: 4:	1	STRUCT	3
5:		Q	$< bn >$
6:		REF	5
7:		6	x
Q: 8:	5	STRUCT	7

Figure 9.3: Two declarers identified

	syn	head	tail
1:		P	$< bn >$
2:		REF	1
3:		2	x
P: 4:	1	STRUCT	3
5:	1	Q	$< bn >$
6:	2	REF	5
7:	3	6	x
Q: 8:	1	STRUCT	7

Figure 9.4: Two declarers equivalenced

```
ACTION determine type equivalences:
   determine type equivalences (1).

ACTION determine type equivalences (>I):
   equal (I, PDECL);
   has synonym (I, S),
      determine type equivalences (I + 1);
   determine type equivalences (I, I - 1),
      determine type equivalences (I + 1).
```

The declarer with index i is compared to all declarers with lower indices, for which the equivalences have already been established.

```
ACTION determine type equivalences (>I, >J):
   equal (J, 1);
   equivalent (I, J);
   determine type equivalences (I, J - 1).

PRED equivalent (>D1, >D2):
   equal (D1, D2);
   has synonym (D1, S),
     (equivalent (S, D2); -);
   has synonym (D2, S),
     (equivalent (D1, S); -);
   [ D1 -> indicant NAME BN],
      postulate (D1, D2),
         get type (NAME, BN, D),
            (equivalent (D, D2);
```

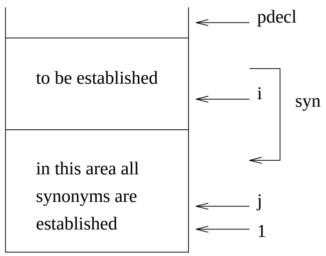


Figure 9.5: Comparing infinite trees

```
          unpostulate (D1, D2), -);
   [ D2 -> indicant NAME BN],
      ( equivalent (D2, D1); -);
   same tree (D1, D2).

PRED same tree (>D1, >D2):
   [ D1 -> primitive PRIM ]
      [ D2 -> primitive PRIM1 ],
         [ PRIM = PRIM1 ];
   [ D1 -> reference to DEC ],
      [ D2 -> reference to DEC1 ],
         equivalent (DEC, DEC1);
   [ D1 -> structured with FIELDS ],
      [ D2 -> structured with FIELDS1 ],
         equivalent (FIELDS, FIELDS1);
   [ D1 -> and DEC DECS ],
      [ D2 -> and DEC1 DECS1 ],
         equivalent (DEC, DEC1),
            equivalent (DECS, DECS1);
   [ D1 -> nil ],
      [ D2 -> nil ].
```

The efficiency of the algorithm can still be markedly improved by exploiting the properties of the representation described above.

9.4 Zosel algorithm

The Zosel algorithm is designed for deciding the equivalence or nonequivalence of a whole set of types simultaneously.
The idea is as follows: the types are partitioned into equivalence classes such that assuming the members of a class to be equivalent does not lead to a contradiction. The input to the algorithm is a set of type definitions of the form

```
<typename> = <declarer>
```

This set is considered as a CF grammar, with the typenames as nonterminals and the declarers as right hand sides.

- Step 1:

Bring all type definitions into Greibach Normal Form. [1]

The declarers are brought into one of the following forms (adding further type definitions with artificial typenames as necessary):

a)	INT	primitive types
	REAL	
	BOOL	
	CHAR	
	VOID	for reasons of symmetry
b)	REF AMODE	possibly introducing a rule for AMODE
c)	ROW n AMODE	for n-dimensional arrays
d)	UNION (M1 M2 ... Mn)	possibly adding rules for Mi
e)	STRUCT (F1 F2 ... Fn)	possibly adding rules Fi = tag$i$ Mi
f)	PROC VOID	
	PROC M	possibly adding a rule for M
	PROC (P1 P2 ... Pn) VOID	possibly adding rules for Pi
	PROC (P1 P2 ... Pn) M	possibly adding rules for Pi and M
g)	LONG n M	

Step 2:
Form equivalence classes as follows:

2.1 Separate the nonterminal symbols into equivalence classes according to $M_i \equiv_1 M_j \Longleftrightarrow$ declarer$_i$ starts with the same symbol as declarer$_j$

2.2 For all non-union classes [2] separate further $M_i \equiv_2 M_j \Longleftrightarrow$ declarer$_i$ same length as declarer$_j$ and $M_i \equiv_1 M_j$

2.3 For all classes separate further $M_i \equiv_3 M_j \Longleftrightarrow$ typenames in declarer$_i$ belong to the same class as the corresponding typenames in declarer$_j$ and $M_i \equiv_2 M_j$.

2.4 If no subdivisions occured in 2.3, then stop, else repeat 2.3.

9.4.1 Example of Z-algorithm

Consider the declarations

```
MODE M = REF REF REF REF REF INT,
     N = REF INT
```

We apply Zosel's algorithm. From Step 1, we obtain the following equations:

```
INT = INT
M = REF M$_1$
M$_1$ = REF M$_2$
M$_2$ = REF M$_3$
M$_3$ = REF M$_4$
M$_4$ = REF INT
N = REF INT
```

Initially, we have only one equivalence class of typenames:

$$\{INT, M, M_1, M_2, M_3, M_4, N\}$$

[1] See section 3.2.3: A CF grammar G is in GNF $\leftrightarrow$ all alternatives in G start with a terminal symbol
[2] special treatment for unions

After performing step 2.1 (splitting on first symbol)

$$\{INT\}\{M, M_1, M_2, M_3, M_4, N\}$$

After performing step 2.2 (splitting on length) no change.
Repeatedly performing step 2.3 (splitting on non-equivalence) gives

$$\{INT\}\{M, M_1, M_2, M_3\}\{M_4, N\}$$

$$\{INT\}\{M, M_1, M_2\}\{M_3\}\{M_4, N\}$$

$$\{INT\}\{M, M_1\}\{M_2\}\{M_3\}\{M_4, N\}$$

$$\{INT\}\{M\}\{M_1\}\{M_2\}\{M_3\}\{M_4, N\}$$

and the partitioning is stable.

9.5 exercises

1. A type declaration can be seen as a CF rule with exactly one, possibly infinite, terminal production. Calling this class of grammar CF1*, is $p_1, p_2 \in CF1 * L(p_1) = L(p_2)$ decideable?

2. Prove the equivalence of

   ```
   MODE P = STRUCT (REF P x),
        Q = STRUCT (REF Q x)
   ```

   by means of the Z-algorithm.

3. Prove termination of the K-algorithm. What is the complexity of the K-algorithm?

4. Prove termination of the Z-algorithm. What is the complexity of the Z-algorithm?