CDL3

compiler description language

CDL3 manual

Version 1.2.7, February 27, 2008

Cornelis H. A. Koster
Paul A. Jones
Marc Seutter
Informatics department
Nijmegen University
Nijmegen
The Netherlands

Jean G. Beney
Informatics department
INSA de Lyon
Villeurbanne
France

Introduction

This is the working version of the manual for the Compiler Description Language CDL3 and its development system. This language represents the third step in an evolution process which included the CDL1 language with its compiler-compiler [KOS74b] and the CDL2 language with its CDL2 LAB[BAY81]. CDL3 [KB91] is an implementation language for compilers, balancing on the borderline between grammars (in which it is easy to construct a parser) and implementation languages (in which large software systems like compliers can be implemented securely and efficiently).

In the computing community the term **implementation** stands for a major part of the software development cycle, starting after the design phase and ending with the first delivery of the product. Implementing large software systems is the honourable battlefield of the professional informatician, the serious side of programming.

Implementing has the same relationship to freshman programming as war to manœuvers: professionalism replaces enthusiasm, collaboration is more important than individual accomplishment, harsh exigencies come in the place of academic leisure. Even though it is a peaceful and constructive activity, implementation projects have been known to end in disaster, in the brutal suppression of promise and in slavery.

Implementation is the technical side of software engineering. It is a human activity and therefore formal questions are often completely overshadowed by pragmatic considerations. In implementation it is not important what is possible, but what is feasible with the given human and material resources and in the time allowed.

It has long been realised that one of the important contributions to the success or failure of an implementation project comes from the choice of the programming language used and its support on the development hardware. By itself, the choice of a suitable implementation language cannot guarantee success but an unsuitable language can lead to a long and costly implementation, resulting in an unstable product, which is expensive to maintain and has a short economic life.

The implementation language is not a panacea but an important tool, that should be chosen with care.

CDL3 is an implementation language based on affix grammars. It rides the borderline between syntactic formalism and programming language, and tries to combine the good properties of both.

The control structure and data structures have been choosen such that it is extremely easy to write deterministic parsers and transducers in CDL3. In this sense, CDL3 is a Compiler Description Language (hence the acronym). Its applicability is, however, not limited to compiler construction. The language is wellsuited, more in general, for all applications that can be characterized as *syntax-directed*: transduction between well-defined formalisms, communication between processes (human and machine) adhering to well-established protocols, or interpreter-like systems, interactively obeying a set of commands.

In this manual, we introduce CDL3 as a programming language and show by small examples how to use it as a Software Engineering tool. Due to its powerful datastructures, strong typing and helpful development tools, it provides a fast and secure approach to the development of large C programs.

The manual is also a textbook on the basic concepts of programming and programming languages. As such it is hoped to have a usefulness not limited to the implementation of compilers in CDL3.

Chapter 1

Algorithms

In this chapter we will deal with the algorithmic structure in-the-small of CDL3 programs, on the algorithmic side. (The object and type aspect is described in the next chapter, chapter 3 describes the structure-in-the-large). Before that, we will introduce some concepts and terminology pertaining to programming languages in general, which is intended to show the motivation for the particular mechanisms present in CDL. In a number of places we give examples using a representative high-level language. No deep knowledge of that language is required.

1.1 Concepts and terminology

This section is concerned with elements of a philosophy of programming languages rather than with details of CDL.

Consider some program, represented as a piece of text. This text is not an amorphous mass of symbols but consists of a hierarchy of constructs according to the syntax of the programming language. These constructs can be broadly classified into those denoting **algorithms** and those denoting **objects** and **types**. Other constructs appearing in the syntax of the language can be considered as ancillary constructions in describing the relationship between the algorithms involved in the program.

1.1.1 Entities

The dichotomy between algorithms on the one hand and objects and types on the other is similar to the distinction between verbs and nouns in natural languages. It may be an artifact of our way of thinking about the world or it may be its very basis. At any rate it is so helpful in thinking about algorithms and so widely spread amongst programming languages that we will turn it into dogma: the meaningful parts of any program are those that can be classified into algorithms, objects and types. In this classification it is at first sight somewhat disturbing to find objects and types (as classes of objects) to be discussed in one breath but every object is an instance of some type, and many relevant properties of individual objects are attributes of their types, and vice versa. At this point in the present discussion there is no need to distinguish between the two. It is awkward that we do not possess one term to denote both objects and types, but we will have to live with that.

Algorithms, objects and types are pieces of program text, that have two important properties:

- They either have a **name** or can be given one
- Upon execution of the program they *possess* some **value** internal to the computer:
  - an algorithm possesses "executable code"
  - an object possesses the internal representation of some value
  - a type possesses a class of internal representations for its instances and a class of operations applicable to them.

Algorithms, objects and types together we will call *entities*. The fact that entities can have a name proves to be one of the keys to successful programming, **abstraction**. The most convenient way to denote an entity is by its name. Indirectly this allows us to manipulate values during execution of the program.

1.1.2 Composed and elementary entities

An entity occurring at a specific place in a program text can be either an **elementary entity** or a **composed entity**.

We call an entity *elementary* if, according to the syntax and semantics of the language and at the level of abstraction provided by the context, it cannot sensibly be decomposed into other entities.

Thus the identifier pi bears no relationship to the identifiers p and i: it is the name of an elementary object, viz. a named constant. The PASCAL expression

$$\textbf{if } a > b \textbf{ then } a \textbf{ else } b$$

on the other hand is a composed algorithm for computing the maximum of a and b.

1.1.3 Abstract and concrete entities

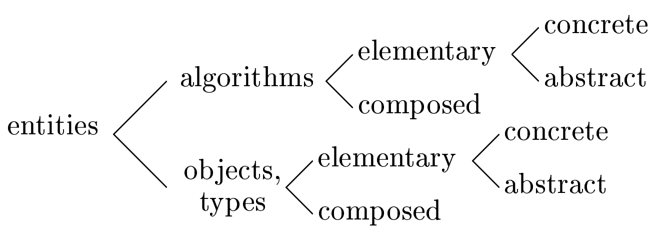


Figure 1.1: Taxonomy of constructs

For any programming language, we will call those elementary entities that are part of the language the *concrete entities* of that language. The programmer is generally free to add more entities to the language (by declaring them) which will then be called *abstract entities*: abstract entities are those that the programmer has to define himself, concrete entities are those defined for him and available for his use. These relations are shown in figure 1.1.

1.1.4 Kernel of the language

The kernel of a programming language comprises

- its **elementary concrete algorithms** (such as: assignation, jump, selection, subscription, coercions, algebraic operations and functions, input/output commands)
- its **elementary concrete types** (highly dependent on the language, these may be denoted by constructions like "`int`" or "1..32767" or "`BIN FIXED (31)`")
- its **elementary objects** (in this category falls e.g. a predefined constant "*pi*").

1.1.5 Construction mechanisms

We use the term *construction mechanisms* for the mechanisms in the language that serve to build composed constructs out of simpler ones, viz.

- for algorithms: **control structures** (the canonical Dijkstra collection (**while**, **if** and **;**) or e.g. LISP conditionals) as well as the **function application** or **procedure call** notation.
- for objects and types: **data structures**, e.g. (in ALGOL 68) "**ref m**" or (in PASCAL) "**record** ... **end**" or "**array**[1..*n*] **of** *integer*" as well as the associated **value constructors** (e.g. in ALGOL 68: row displays and structure displays).

1.1.6 Abstraction mechanisms

Abstraction mechanisms serve to build new elementary constructs out of constructs in the language. They usually comprise:

- declarations for algorithms (in the form of procedures, functions, subroutines, operators or macros)
- declarations for objects (such as variables, constants and arrays)
- declarations for types (abstract data types)

1.1.7 Extension mechanisms

Extension mechanisms serve to borrow new elementary constructs from outside the language. Classically they comprise libraries and macros, or even assembly code patches. Notice that they generally serve for **semantic extension** only. The notion of **syntactic extension**, popular in the early sixties, was less fruitful and seems to have been largely abandoned.

A language providing built-in extension mechanisms is called an **open-ended language**, due to the open-ended character of its semantics.

1.1.8 Mechanisms in CDL

The abstraction mechanisms of CDL3 are based on two-level grammars, where the first level provides construction and abstraction mechanisms for algorithms and the second level for objects and types.

In distinction to its predecessors, CDL3 is not a particularly open-ended language. Its kernel comprizes integers, texts and trees composed over them, together with their operations, as well as input/output via the C-library.

CDL was designed deliberately to explore an extreme position. One consequence is the fact that, in learning CDL, the familiar concepts of programming languages and of systematic programming have to be carefully reconsidered. The extreme position taken has proved to be fruitful in simultaneously achieving portability and efficiency [STA80].

In the rest of this chapter we shall deal with CDL3's construction and abstraction mechanisms for algorithms. The next chapter will describe the mechanisms for dealing with objects and types, as well as the kernel of the language.

1.2 Control structures

Apart from their unusually short notation, the **control structures** of CDL3 are rather classical. They are: sequencing, sequential choice and grouping. It will become apparent that the notation for the control structures is quite similar to that in PROLOG. This is no accident, since both languages have as their origin a syntactic formalism. Indeed, it might be said that CDL3 is a deterministic, strongly typed (and therefore highly efficient) variant of PROLOG.

1.2.1 Sequencing

The **sequential execution** of algorithms is indicated by writing a comma between their calls, which can be read: "and then". As an example, the text

```
read a number, print it
```

strongly suggests that first a number is read, and then it is printed. In the same way,

```
read a number, read another number, add them, print sum
```

indicates four algorithms to be executed in strict order. (Of course these are abstract algorithms, they are much too specialized to be included in any concrete programming language).

1.2.2 The conditional

The **sequential choice** between two alternatives is expressed by writing a semicolon between them. This separator can be pronounced "or else". As an example

```
is minus, read number, invert it, print it;
read number, print it
```

might be part of some desk calculator or interpreter. The idea is that `is minus` is the name of an algorithm that may either succeed (if it recognizes a minus sign in the input) or fail (if it doesn't). If `is minus` succeeds, the first alternative is taken, and otherwise the second is taken.

1.2.3 Classification of algorithms

In CDL, all algorithms are classified into categories, depending on their influence on the execution:

- those that can either succeed or fail viz. the **predicates** and **tests**. They can be seen as Boolean procedures whose value is used to control the execution.

- those that cannot fail viz. the **actions** and **functions**. They can be seen as Boolean procedures that always return the value true.

We will meet with tests and functions later and for the time being will only deal with actions and predicates without parameters. We will, in our examples, distinguish actions from predicates by their names: names of predicates will start with `is`.

In the previous example then, `is minus` is a predicate whereas all others are (as their names suggest) actions. They have an effect on either the input (reading or recognizing) or output (printing).

1.2.4 Some syntax and semantics

The body of a procedure in CDL has the form of a **group**. Informally and rather incompletely, its syntax and semantics can be described as follows:

A group consists of one or more **alternatives** separated by semicolons. An alternative consists of zero or more **members** separated by commas. An alternative containing zero members is termed an **empty alternative**. Only the last alternative of a group may be empty. An empty alternative is indicated by a plus sign (**success operator**).

A member can be the (possibly parameterized) call of an algorithm or a guard. The **last member** of an alternative can also be the success operator, the abort operator, the failure operator or an **enclosed group**, i.e. a group enclosed between brackets.

The execution of an algorithm may have an effect on the values of variables and it will either succeed or fail.

A group is executed by executing its alternatives in textual order until either an alternative succeeds or the last alternative has been executed. The group succeeds as soon as an alternative succeeds, and fails if all alternatives fail.

An alternative is executed by executing its members in textual order until either a member fails, in which case the alternative fails, or until all its members have been executed successfully, upon which the alternative succeeds. An empty alternative always succeeds.

The execution of a member consisting of a (parameterized) algorithm call is the execution of that algorithm (with those parameters). The execution of a member consisting of an enclosed group is the execution of that group.

The execution of a guard computes values for certain variables. A call or a guard may either succeed or fail depending on the values of its parameters and, in case it succeeds, may have an effect on its parameters; in case it fails, it has none.

1.2.4.1 Example

As an example, we discuss the body of some interpreter for monadic expressions of the form

$$[+|-]\ \textit{number}$$

which might be programmed

```
is minus, read number, invert it, print it;
is plus, read number, print it;
is number, print it;
report error
```

Notice that `is number` is a predicate: it tries to recognize a number. On the other hand `read number` is an action (which is programmed somewhere else, probably in terms of `is number`): it will rather die than admit to not finding a number. We will program it later.

Notice also that this group as a whole cannot fail: if neither a leading plus sign nor a leading minus sign nor a number is present, it will report an error, and thus succeed. As such, it can be the body of some action.

1.2.5 Enclosed group

The fact that the last member of an alternative may be an **enclosed group** allows the nesting of alternatives in a hierarchical fashion, e.g.:

```
read number,
   (is plus, read number, add them, print result;
    is minus, read number, subtract them, print result;
    report missing operator)
```

This suggests an interpreter for dyadic expressions of the form

$$number\ \{+|-\}\ number$$

whose control graph is shown in figure 1.2.

By the **control graph** we mean a graph depicting the possible flow of control through the members of a group.

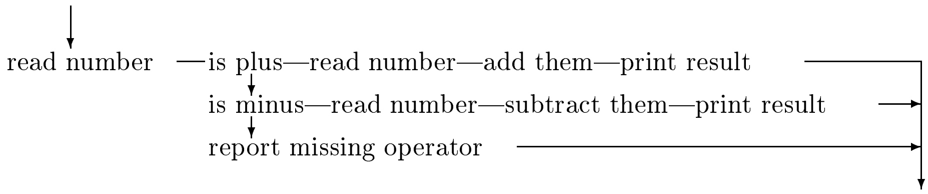


Figure 1.2: Control graph for dyadic expression

After the action `read number` is called, there are three possibilities: either a plus is present, a minus is present, or an error is reported.

Notice that only the *last* member of an alternative can be split in this way, it is not applicable to preceding members, which have to be calls of algorithms or guards. Thus, a closing bracket can not be followed by a comma in a CDL program.

Since the grouping structure can be used again within the enclosed group, the control graph can be nested to an arbitrary depth. Due to the restriction that splitting can occur only at the end of an alternative, the resulting control graph is still very well comprehensible, especially if suitable layout conventions are observed. The resulting control structure is simple and powerful, and supports the strategy:

> Within an alternative, as soon as you feel there is more than one possibility, write a left bracket "`(`" indented on a new line, then program the alternatives one by one before closing the group.

If the term "structured programming" has any meaning, it applies to the resulting programming style [GRU82].

1.2.6 Failure operator

The last member of a group may be a **failure operator**, written as a minus sign. Its execution results in the failure of the current and of the folowing alternatives. Then the current alternative fails.

Notice that (as well as for the abort operator), this does not lead to the execution of the next alternative but to the failure of the actual procedure (or the whole program). These facts are taken into account in the consistency check.

1.2.7 Abort operator

The last member of an alternative may be an **abort operator**, written as a question mark. Its execution results in termination of the program. The abort operator is meant for extricating the program from a hopeless situation, after having suitably reported on that situation and having mended it as far as possible, by returning control to the operating system. The program has no further chance to do anything whatsoever.

This may be the right thing to do in case a more subtle reaction is not possible, e.g. for a compiler that runs out of space.

1.3 Procedure declarations

A **procedure declaration** serves to bind a name to a group and equip it with parameters. It consists of a **procedure heading** and a **procedure body**, separated by a colon and followed by a period.

The heading starts with the type and the name of the procedure. The body is a group.

As an example, we can declare an action `read number`, assuming the existence of `is number`, as follows

```
ACTION read number:
    is number;
    report error.
```

Notice that `read number` cannot fail since `report error` is an action. (Whether it does something sensible is another matter.)

Notice that the type of the procedure is not a name, but one of a small set of magic words, written in capital letters.

We can now embark on the complete definition of a desk calculator recognizing

$$[\textit{number}\ [\ \{+|-\}\ \textit{number}]^{*}\text{end-of-line}\ ]^{*}\ \textit{end-of-file}$$

We will proceed in a Top-Down fashion, first postulating abstract algorithms and then declaring them as we go along. We will describe only the algorithms and assume that the values found are manipulated implicitly e.g. as global objects. We will leave the consideration of objects until the next chapter.

```
ACTION desk calculator:
    is number, rest after number, desk calculator;
    is end of file;
    complain and skip line, desk calculator.
```

```
ACTION rest after number:
    is plus, read number, add them, rest after number;
    is minus, read number, subtract them, rest after number;
    is end of line, print result;
    complain and skip line.
```

We assume the availability of suitable predicates for recognizing plus and minus signs, digits and end-of-file, as well as actions for arithmetic and printing.

Bridging the gap, we program:

```
ACTION read number:
    is number;
    report number missing, assume zero.
```

A number is a nonempty sequence of digits *digit digit**, which can be expressed as

```
PRED is number:
    is digit, other digits.

ACTION other digits:
    is digit, other digits;
    + .
```

This example is still quite unsatisfactory, because all computations take place as implied side-effects. At this point it has become evident that we must introduce mechanisms for manipulating objects, and for communication between algorithms.

1.3.1 Consistency of algorithms

The differentiation between algorithms that can or cannot fail, leads to consistency rules.

In a group, an alternative is executed when the previous one fails. Thus each alternative that is not the last of a group must contain a test or a predicate in order to be able to fail. A group can only fail if its last alternative contains a test or a predicate. When a procedure is declared as being a test or a predicate, its body must be a group that can fail. A function or action must have a body that cannot fail.

CDL3 differs from most programming languages in providing a tight consistency check of programs – see Ch. 8.

1.4 Communication between algorithms

During the execution of a program, it must be possible to communicate information between algorithms. Generally speaking, programming languages provide three mechanisms for such communication:

- input/output, which is slow, ponderous and ill-defined, suitable only for low levels of interaction (or large quantities)
- global variables, with all their concomitant problems of unwanted access and side-effects
- parameters and local variables, the most explicit and best defined mechanism.

In CDL, parameters and local variables can be associated with an algorithm. Due to the roots of this language in *affix grammars* [KOS71b] these are termed *affixes.* This term is also used for the (global) variables, constants and types, which we will describe in the next chapter. Each affix is denoted by a name.

1.4.1 Call with actual parameters

In calling algorithms, **actual parameters** can be supplied. The notation

```
alg ( PAR1 , PAR2 )
```

means that the algorithm `alg` is called with actual parameters `PAR1` and `PAR2`. An actual parameter can be a single affix or an **affix expression**, whose structure we will describe later (see 2.1.2).

1.4.2 Formal parameters

In CDL, **formal parameters** must be specified according to their *direction.* Three directions are distinguished:

- **input parameters**, e.g. `alg ( >INPAR ) :`

  Obviously, `INPAR` goes in.
- **output parameters**, e.g. `alg ( OUTPAR> ) :`

  Here, `OUTPAR` goes out.
- **transient parameters**, e.g. `alg ( >TRANSPAR> ) :`

  `TRANSPAR` goes both in and out.

Input parameters are passed *by-value*, i.e. upon entering the algorithm, each input parameter obtains a copy of the value of the corresponding actual parameter. In the body, the input parameter is not supposed to be modified, but it is a local variable.

Output parameters are passed *by-result*, they are local variables whose values are copied to the corresponding actual parameters upon leaving the algorithm successfully. Upon failure, they are not copied.

Transient parameters are passed both by-value and by-result, in the sense given above. This parameter mechanism resembles that of ALGOL W, rather than that of PASCAL.

The reasons for preferring call by value/result over **call by-reference** or **call by-name** should be obvious: in an implementation language, the decoupling of calling and called side is desirable in order to control side effects. Furthermore, (static) aliasing problem is impossible. Thus, an implementation of CDL3 may use call-by-reference, if the hardware makes this preferable (has good facilities for indirect addressing).

Language gourmets may further note that CDL3 also avoids the dynamic alias problem by restrictions on composed actual parameters.

1.4.3 Local variables

Every variable occuring in a declaration which is not a **bound variable** (i.e. one of the formal parameters) or a global variable (see 3.3.1) is considered as a **free variable**, local to the declaration. The consistency checks ensure that local variables obtained accidentally through typographic mistakes will be signaled.

1.4.4 Syntax of algorithm-headings

The heading of an algorithm consists of the following elements in order:

- the type of the procedure
- the name of the procedure
- the (optional) formal parameters

An example of a heading containing all those elements is:

```
ACTION enter pair ( >HEAD , >TAIL , REF> )
```

It suggests that a pair (`HEAD` and `TAIL`) goes in, some global data is changed and a `REF` comes out. A suitable heading for a procedure entering a pair into a global list and returning a reference to it.

1.4.5 Example: `is number`

Suppose we already have a

```
PRED is digit ( D> )
```

with the properties:

- if the next symbol of input is indeed a digit
  - the input is advanced by one symbol
  - `D` gets the value of the last digit read, and
  - `is digit` returns true
- if the next symbol of input is not a digit
  - the input is not advanced
  - the value of `D` is not defined, and
  - `is digit` returns false.

We call such an algorithm an **exact recognizer** for digits, because it not only recognizes a digit correctly, but also recognizes a non-digit as such without any effect on the input.

We can use it to build an exact recognizer for numbers, which computes the value of the number as a side-effect while recognizing the number.

```
PRED is number ( VAL> ) :
   is digit ( VAL ),
      other digits ( VAL ).

ACTION other digits ( >VAL> ):
   is digit ( DIG ),
      append digit to val,
         other digits ( VAL );
   + .
```

By itself this is a rather simple piece of program, but it contains one fuzzy spot: how can we append a digit to `VAL`? We must perform some (elementary) calculation and we need yet another mechanism that is called guard.

1.5 Guards

The **guards** are the elementary algorithms of CDL3. They are confrontations between variables (see 2.1.6) and expressions (see 2.1.2). A guard is enclosed between square brackets. It expresses a condition that must be satisfied between the values of variables and constants.

In order to satisfy this condition, the execution of a guard can give to a variable a value that is computed (**join guard**) or just taken from another variable (**assign guard**). It can also compare the value of a variable with the value of another variable (**equal guard**) and fail if they are not equal, or compare the value of a variable with a constant (**split guard**) and fail if they are not equal.

1.5.1 Arithmetic guards

By *arithmetic guards* we mean guards involving only affixes whose domain is the whole numbers (**integers**). These provide a notation for arithmetic operations.

As an example `append digit to val` can be replaced by the join guard:

```
[ 10 * VAL + DIG -> VAL ]
```

Completing the previous example, we obtain:

```
ACTION other digits ( >VAL> ):
   is digit ( DIG ),
      [ 10 * VAL + DIG -> VAL ],
         other digits ( VAL );
      + .
```

Examples of guards of the other kinds:

- assign the value of `DIG` to `VAL`:

  ```
  [ DIG -> VAL ]
  ```

- compare `DIG` and `VAL`:

  ```
  [ VAL = DIG ]
  ```

  The comparison works only between variables, comparing their instantanious values.

- compare `DIG` and `9`:

  ```
  [ DIG -> 9 ]
  ```

  This is actually a *split*. In the general case, the value of a variable is split into its components.

In the next chapter, we will describe the type system of CDL3 and say more about guards.

1.6 Classification of algorithms revisited

An algorithm has two (independent) attributes

- whether it returns a *result* (does it always succeed, or may it also fail)
- whether it has a global *effect*

By a **global effect** of an algoritm is meant "a change to the observable universe upon its successful execution". The philosophy of effects will later be described more precisely. For the moment we will resort to the reader's intuition, defining a global effect as: "a side effect otherwise than through a parameter".

Crossing these two attributes leads to four types of algorithms:

specification	can fail	has a global effect
`TEST`	yes	no
`PREDICATE`	yes	yes
`ACTION`	no	yes
`FUNCTION`	no	no

Let's give some examples of headings of algorithms illustrating each type. They are all **abstract algorithms** because they do not belong to the small set of concrete algorithms built into the laguage, but can be expressed in term of them.

- `TEST equal ( >X , >Y )`
  to test for numerical equality.
- `TEST is small letter ( >X )`
  to determine whether the value of `X` is the code of a letter in lower case (e.g. in ASCII: between 97 and 122).
- `PREDICATE is number ( VAL> )`
  an exact recognizer for numbers, computing their value. It has an effect on the input.
- `PREDICATE can pop ( X> )`
  returns false if the global stack is empty; otherwise it pops an element, assigning its value to `X` and returns true. The effect is on the global stack.
- `ACTION push ( >S , >X )`
  aborts if the stack `S` is already full; otherwise, pushes `X` onto `S`.
- `ACTION print int ( >X , >WIDTH )`
  prints `X` in `WIDTH` positions. The effect is on the output file.

The fourth type, `FUNCTION`, may at first seem superfluous: no value returned, no global effect. One possible application is to achieve effects that the programmer deems not to form part of the process itself, e.g. tracing

- `FUNCTION trace ( >X )`
  the current value of `X` is printed on a trace file.

The major application is, however, to achieve effects solely through the parameters, as is usually the case in computations:

- `FUNCTION add ( >A , >B , C> )`
  an addition in three-address style.

- `FUNCTION make ( Y> , >X )`
  assign `X` to `Y`.

Notice that these two functions can be expressed directly by guards.

The programmer has some degree of freedom in specifying the type of his algorithms, but is well advised to specify as exactly as possible: The specifications form the basis for a *semantic check* on side effects, checking what the programmer says he wants to do against what he actually does [FEU78]. More about this in Chapter 8 on static semantic checks.

1.7 Exercise

The following exercise is intended to test the reader's familiarity with the notation introduced, in particular the control structures.

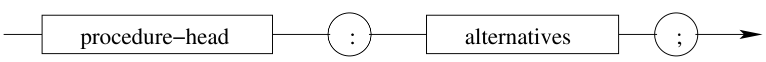
Consider the problem of filling a pot with marbles. We may put another marble into the pot by means of the action **`add a marble`**, and test whether it is full by one of the tests **`pot is full`** or **`pot is not full`**. Now write actions **`fill the pot`**, according to each of the two following schemas (expressed in ELAN):

1. ```
   PROC fill the pot:
      WHILE pot is not full
      REP
         add a marble
      ENDREP
   ENDPROC;
   ```
2. ```
   PROC fill the pot:
      REP
         add a marble
      UNTIL pot is full
      ENDREP
   ENDPROC;
   ```

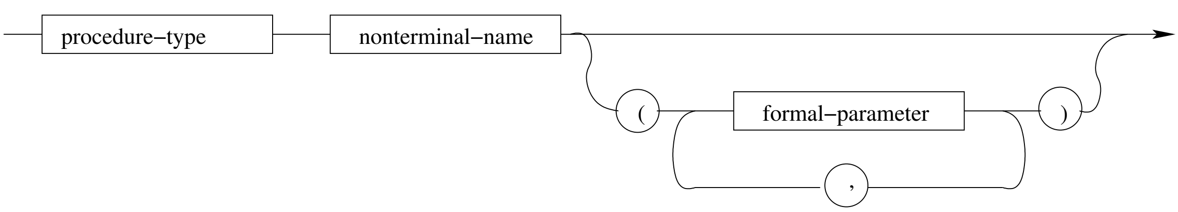
1.8 Syntax summary

We summarize the syntax introduced in this chapter.

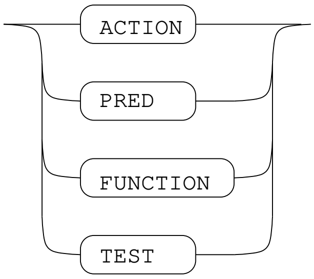
procedure



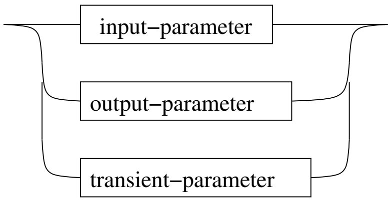
procedure–head



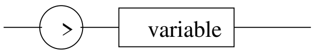
procedure−type



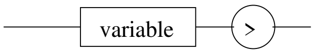
formal−parameter



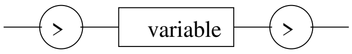
input−parameter



output−parameter

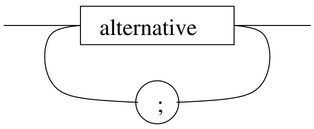


transient−parameter

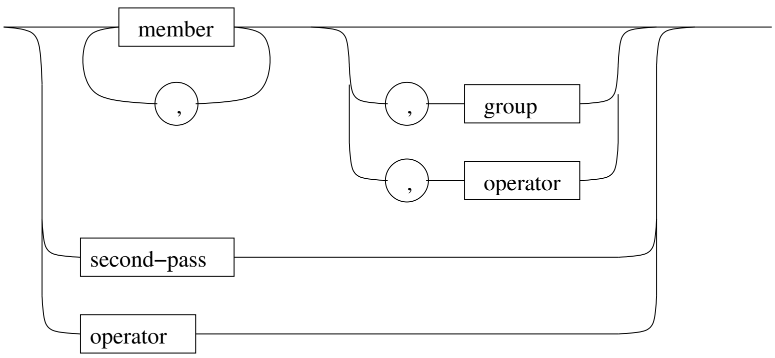


The following rule is simplified: see 7

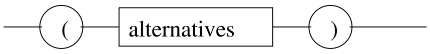
alternatives



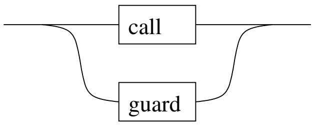
alternative



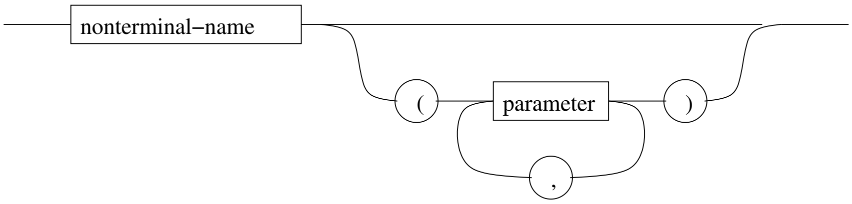
group



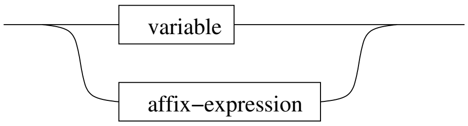
member



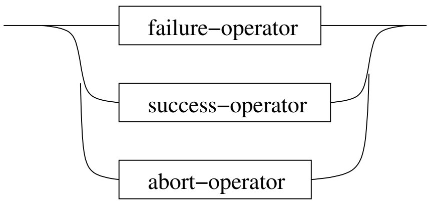
call



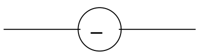
parameter



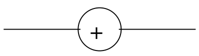
operator



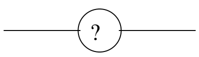
failure−operator



success−operator

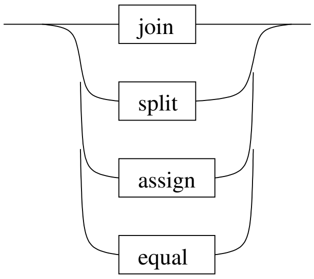


abort−operator

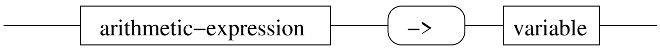


guard

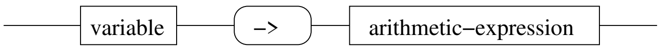
confrontation



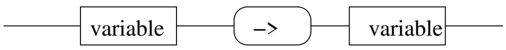
join



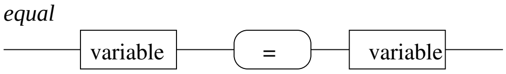
split



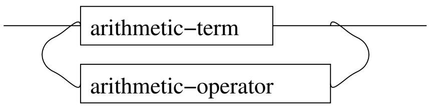
assign



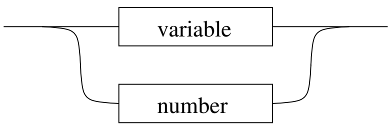
equal



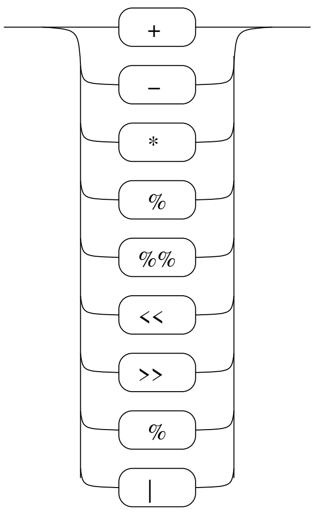
arithmetic− expression



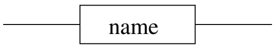
arithmetic−term



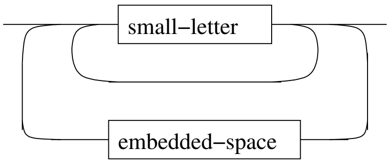
arithmetic−operator



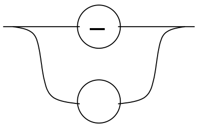
name



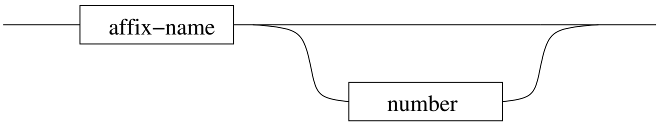
nonterminal−name



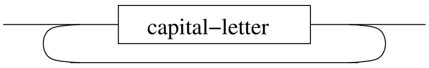
embedded−space



variable



affix−name



Chapter 2

Objects and types

CDL3 provides the means to declare types, whose domains are tree structures, as well as variables and constants of these types and the operations to manipulate them.

2.1 Defining affixes

2.1.1 Affixes

A **nonterminal affix**, composed of capital letters, denotes a type. Furthermore it may serve as an **affix variable**, standing (as a name) for some object of that type. In the latter case it may optionally be followed by a number to distinguish different instances of the nonterminal affix, eg `EXPR` and `EXPR1`.

A **terminal affix** denotes a constant value of some type. It consists of small letters. Spaces may be used to enhance readability.

A terminological remark: in the sequel, when there is no ambiguity (for example when there is another adjective), we will simply use the word **affix** for a nonterminal or a terminal affix.

2.1.2 Affix expressions

An **affix expression** is a sequence of affix variables and terminal affixes. It serves to denote a tree pattern according to the metarules. Each affix variable denotes a subtree of the corresponding type, and a terminal affix denotes a leaf.

Affix expressions are used to define possible tree patterns in a type definition or to represent effective tree structures in the algorithms. In the latter case, it can also contain arithmetic and text expressions.

2.1.3 Metarules

Every affix must be defined by a **metarule**, which consists of the nonterminal affix to be defined followed by a double colon, a list of meta-alternatives separated by semicolons and followed by a period. A **meta-alternative** is an affix expression. The semi-colons can be read as “or”.

A metarule defines a **domain** which is the union of the domains of its alternatives. The domain of an alternative is the collection of tree structures *conforming to it* (2.1.5).

Examples

A list of names can be defined by:

```
LIST :: empty ; NAME LIST .
```

and a nonempty list by:

```
LISTP :: conc NAME LISTP ; NAME .
```

Notice that the **marker** `conc` is redundant, it merely serves to enhance readability.

In the next sections, we will develop a fragment of a translator of arithmetic expressions. Notice that, in this example, the term *expression* will concern the translator being developed, while the term *affix expression* will concern CDL3.

In our example, the abstract syntax tree (see 6.6) of an expression shall be either a name denoting a variable, or an integer constant, or an expression composed of an operator and two subtrees that are expressions:

```
EXPR :: var NAME ; const INT ; EXPR OP EXPR .
```

Notice that in each of these examples the last alternative has no marker.

```
OP :: plus  ; minus ; mult ; div .
```

defines `OP` as an enumeration of alternatives that consists only of a marker.

We have now introduced the affixes `LIST`, `LISTP`, `EXPR` and `OP` which can be seen as four different types.

2.1.4 Synonyms

In order to enhance readability, an affix can have a number of **synonyms**. They may be stated at the beginning of a metarule, separated by commas. They may also be declared by a **synonym rule**, that is an affix rule with the symbol `=` instead of `::`.

Furthermore synonyms can be declared implicitly: an affix variable consisting of an affix nonterminal followed by a number is a synonym for that affix nonterminal.

Examples

We can introduce `E` as a synonym for `EXPR` by:

```
E, EXPR :: var NAME ; const INT ; EXPR OP EXPR .
```

or by:

```
EXPR :: var NAME ; const INT ; EXPR OP EXPR .
E = EXPR .
```

`E1` and `E1789` are implicit synonyms of `E`, therefore of `EXPR`.

This mechanism can in particular be used to make "sugared" versions of the predefined types `INT` and `TEXT`. As an example, `COUNTER = INT.` introduces a name for integers to be used as counters.

Notice that the rule:

```
E :: EXPR .
```

does not define a synonym but defines `E1` as being a new type whose domain happens to be the same as that of `EXPR` (**strong abstraction**).

2.1.5 Conforming affix expressions

An affix expression is said to **conform** to a given affix if that affix expression has the same structure as an alternative of the metarule that defines the affix or a synonymous affix. It means that there must be the same terminal affixes in the same places and the same affixes or synonymous affixes (or conforming subexpressions, see 2.2.2) in place of the nonterminal affixes

Example

`E1 OP0 E2` conforms to `EXPR`.
`const 0 plus var "x"` conforms to `EXPR`.

2.1.6 Variables

In an algorithm declaration, an affix denotes a **variable**, i.e. an object of that type which can receive a value. The set of possible values of a variable is given by the corresponding metarule. There are global variables whose scope is a module (see 3.3.1), and formal parameters and local variables whose scope is the rule where they are declared.

2.2 Guards

In order to build trees and to travel through them, we will use **guards**. In those guards, we use affix expressions that describe the structure of the trees that we want to build or the expected structure of the tree that we traverse.

2.2.1 Classification of guards

There are four sorts of tree guards: *join, split, equal* and *assign*.

2.2.1.1 Join guards

To build a tree we use a **join guard**:

```
[ var NAME -> EXPR ]
```

builds an expression tree that contains only a given variable name.

```
[ EXPR1 OP EXPR2 -> EXPR ]
```

builds an expression tree with an operator and two subexpressions.

The affix expression must *conform* to one of the alternatives for the affix on the right hand side of the join guard. All variables occuring in the left hand side of the join guard must have already a value so that the tree that is built will be completely known. Then the guard always succeeds and as it has no hidden side effects, it is a function.

2.2.1.2 Split guards

To explore a tree we use a **split guard**. This guard confronts the actual value of a variable with an affix expression.

Examples

```
[ EXPR -> const INT ]
```

looks whether the tree which is the value of `EXPR` contains only a constant and assign the value of that constant to the variable `INT`. The guard fails when the tree has a different structure.

```
[ EXPR -> EXPR1 OP EXPR2 ]
```

looks whether the tree value of `EXPR` is an operator expression and assigns the values of its components to `EXPR1, OP` and `EXPR2` respectively. The guard fails if the tree has another structure.

Since the guard can fail when a tree has several possible structures, it is a test. When a tree has only one possible structure, the guard is a function. This is the case in the affix rule:

```
VARDEF :: NAME TYPE .
```

The following split guard is a function:

```
[ VARDEF -> NAME1 TYPE1 ]
```

The guard has no chance to succeed if the affix expression has not a possible structure for the variable. Then the CDL3 compiler will report an error if the affix expression does not conform to the affix.

2.2.1.3 Assign guards

The **assign guard** simply gives to a variable the value of another variable.

Examples

```
[ EXPR -> E0 ]
```

The two variables must be synonymous affixes, and the guard always succeeds.

2.2.1.4 Equal guards

The **equal guard** is used to compare the values of two variables for equality.

Examples

```
[ E0 = EXPR ]
```

This sort of guard is obviously a test. The variables must be synonymous affixes and have already a value.

2.2.2 Subguards

The join and split guards contain an affix and a conforming affix expression. It means that only one affix rule is concerned by a given guard. Then only one tree node is concerned by a guard.

It can be practical to work on several tree nodes in the same guard. This is possible by the use of **subexpressions** that denote **subguards** embedded in a guard.

2.2.2.1 Double join

To build a tree that represents the arithmetic expression `1 + EXPR`, we must build an `EXPR` node `const 1`, an `OP` node `plus` and use them to build another `EXPR` node:

```
[ const 1 plus EXPR -> E0 ]
```

2.2.2.2 Double split

The optimisation of constant expressions may contain a test to see if a given expression is composed of two constants linked by a plus operator. Then the sum of the two integers is computed and gives the simplified expression:

```
[ E -> const INT1 plus const INT2 ] ,
  [ const INT1 + INT2 -> EXPR ]
```

Notice also the embedded arithmetic join:

```
  [ INT1 + INT2 -> INT ]
```

2.2.3 Directions in guards

The variables appearing in a guard can be seen as arguments of predefined elementary algorithms. The direction of the corresponding formal parameters is summarized in the following figure:

	position of the variable in the guard	
sort of guard	left hand side	right hand side
join	input	output
split	input	output
assign	input	output
equal	input	input

In a join or split guard, the variable of the left hand side can also appear in the expression of the right hand side. Then is has to be seen as a transient parameter.

2.2.4 Affix expressions as parameters

It is possible to use an affix expressions (rather than a single affix) as actual parameters. In that case an **implicit join** will be executed just before parameter passing (for an input parameter) or an **implicit split** just after it (for an output parameter), or both for a transient parameter.

Notice that a split implies a test and therefore a call to an action or a function can imply a test!

Examples

Assuming the definition

```
ACTION expression tail(>EXPR0,EXPR>): ... .
```

in the call `expression tail( EXPR0 plus EXPR1 , EXPR )` the evaluation of the input parameter implies a join.

2.2.5 Type consistency

In the course of the preceeding explanations, some type conditions have already appeared:

1. in a split or join guard, the affix expression must conform to the affix;
2. in an assign or equal guard, the affixes must be synonymous.

2.3 Information flow

As seen above, the direction of the parameters in an algorithm call is known as well as the direction of informations in a guard. This allows the compiler to check the coherency of the **affix flow** according to the following CDL philosophy.

2.3.1 Defining/applying positions

An input parameter receives a value at the start of the rule. Similarly, a variable will receive a value when passed as an output argument to a call or in an output position in a guard. Such positions are called **defining positions**.

Similarly, an input argument position and the return from a rule for an output parameter are called **applying positions**.

A transient argument position must be seen as both an applying position and a defining one. A transient formal parameter has a defining position at the start and an applying position on return.

Example

```
ACTION expression ( EXPR>):
   is term (EXPR1), expression tail (EXPR1, EXPR).

ACTION expression tail (>EXPR0, EXPR>):
   is symbol("+") ,
      term(EXPR1) ,
        expression tail( EXPR0 plus EXPR1 , EXPR ) ;
   [ EXPR0 -> EXPR ] .
```

`EXPR0` is defined on the entrance of the rule and applied in the recursive call and in the assign guard;

`EXPR1` is defined on return from `term` and applied in the recursive call;

`EXPR` is defined on return from the recursive call or in the guard, depending on the alternative used; it is applied on return from the rule.

2.3.2 Affix flow rules for local variables

The CDL philosophy for the use of variables is simple: the computation takes place from left to right, i.e. in a rule each local variable or parameter must have, in any alternative where it appears, a defining position before any applying position. That implies that you can not use in an alternative a value computed in a previous alternative. This is the rule of **independency** of alternatives.

Example

The affix flow in the above example can be shown for the first alternative:

```
ACTION expression tail(>EXPR0,EXPR>):

        is symbol("+"),          term(EXPR1),

        expression tail(EXPR0 plus EXPR1,EXPR);

        [ EXPR0 -> EXPR ] .
```

and for the second alternative:

```
ACTION expression tail                  (>EXPR0,EXPR>):

is symbol("+"),  term(EXPR1),

expression tail(EXPR0 plus EXPR1,EXPR);

                                       [ EXPR0 -> EXPR ] .
```

2.3.3 Consistent substitution

The grammatical origin of CDL3 explains the rule that alternatives must be *independent*. In a grammar each alternative is defined independently from the others and a variable has a meaning only for one alternative. Furthermore, a variable has only one meaning in an alternative; for a given call, a variable represents always the same value : that is the **consistent substitution** rule of two-level grammars.

A CDL3-program is L-attributed: the *flow-of-information* within a procedure body is strictly left-to-right. No information can be passed from one (failed) alternative to the next.

A major concern in writing large grammars (as in writing large programs) is the large number of affixes to be passed explicitly from one part of the grammar to others. This communication overhead, resulting from the explicit applicative character of grammars, has to be overcome in order to obtain a practical notation for programming.

That is why we introduced also transient parameters, which may seem to violate the consistent substitution rule. Consider:

ACTION p (>X_1, X_3>):

a (X_1, X_2), b (X_2, X_3).

Purely as a shorthand for such pairs of parameters, winding their way through the calls, we also admit *transient* parameters, written

```
ACTION p (>X>): a(X), b(X).
```

Example

the previous rule can have a transient parameter that carries the tree being built:

```
ACTION expression tail(>EXPR>):
        is symbol("+") ,
          term(EXPR1) ,
            [ EXPR plus EXPR1 -> EXPR ] ,
                expression tail(EXPR) ;
        + .
```

2.3.4 Modifying input parameters

An input parameter can be modified, but as its value is not passed on return from the procedure, this modification will not affect the corresponding argument. Thus an input parameter can be seen as a local variable initialized by the value that is passed when calling the procedure.

2.3.5 Global variables

Similar rules are to be applied to global variables (see 3.3.1): they must get a value before being used, and you must not use in an alternative a value computed in a previous alternative.

The initialisation of global variables is not checked, but the module prelude facilitates its placing (see 3.3.4). The alternative independance is enforced by the prohibition of defects (see 4.2.3): you must not modify a global variable when the actual alternative can still fail.

2.4 Predefined types

2.4.1 Integers

We already introduced in the previous chapter integer variables. The concrete type `INT` might have been defined in term of trees storing the unary representation or of some other representation of the natural numbers. But then the definition of the arithmetic operations would have to be done in term of tree operations and that would not allow us to use the simplicity offered by the hardware. So the type integer is predefined in CDL3 and we can use the usual arithmetic expressions.

Example

To complete the example of the previous chapter, `VAL` and `DIG` have to be defined as synonymous of `INT`.

```
VAL, DIG = INT .
```

2.4.2 Texts

Similarly, the concrete type TEXT could have been defined as a list of characters. But the concatenation would then have been a very slow operation. Therefore there is a predefined type `TEXT` with affix expressions that have a special meaning: an affix expression that contains only text variables and constants separated by plus signs (+) will be executed as a concatenation of the texts.

Example

```
NAME = TEXT .
```

defines an affix that can be used for text variables.

```
[ NAME0 + ".log" -> NAME ]
```

concatenates the value of `NAME0` and the suffix `".log"` and puts the result in `NAME`.

2.5 More on type checks

The **conformity** check of an affix expression versus a nonterminal affix is a **strong type check**. It means that when an affix is expected we must put precisely that affix, a synonym or an expression conforming to that affix.

As a normal meta-rule (::) does not define a symetrical relation, we must declare precisely what we want to allow. With the meta-rules:

```
E, EXPR :: var NAME ; const CONST ; EXPR OP EXPR .
CONST :: INT.
```

`const CONST` conforms to `EXPR` and `const INT` also does because `INT` conforms to `CONST`.

But with the meta-rules:

```
E, EXPR :: var NAME ; const INT ; EXPR OP EXPR .
CONST :: INT.
```

`const INT` conforms to `EXPR` but `const CONST` does not.

In this case it is possible to force `CONST` as being temporaryly considered as a synonym of its right hand-side `INT` by prefixing it by an exclamation point. Then `const !CONST` conforms to `EXPR`.

2.6 Example: Association lists

As an example, we shall show how to define an abstract data type in CDL3.

By an *association list* we mean a set of pairs `<KEY, ELEMENT>`, such a pair indicating that the `ELEMENT` is associated with the `KEY`. A set of pairs is either empty or it is composed of a pair and another set.

This (recursive) definition of association list can be expressed straightforwardly in CDL3:

```
ALIST :: empty ;
        KEY ELEMENT and ALIST.
```

The terminal affix `and` is not really necessary, it has been interpolated in order to enhance the readability of the affix expression for human eyes. It should be noticed that this embellishment is not paid for by extra storage: in the *internal* representation of `ALIST`s it will not be present.

As types for both `KEY` and `ELEMENT` we take texts:

```
KEY, ELEMENT :: TEXT.
```

We now implement the operations necessary to obtain the conventional datatype `ALIST`.

Denotation of the empty list and test for emptyness are particularly simple:

```
FUNCTION empty list(ALIST>):
  [empty->ALIST].

TEST is empty list(>ALIST):
  [ALIST->empty].
```

The access operations are tests, due to the fact that the list may be empty.

```
TEST head(>ALIST,KEY>,ELEMENT>):
  [ALIST-> KEY ELEMENT and ALIST].

TEST tail(>ALIST>):
  [ALIST-> KEY ELEMENT and ALIST].
```

The operation for entering a pair into an association list will

1. add the pair, if no pair with the given key is present, or
2. replace the element associated with this key by the new one, if the key is already present.

It ensures in this way that keys are unique.

```
FUNCTION enter (>ALIST>,>KEY,>ELEMENT):
  [ALIST->KEY1 ELEMENT1 and ALIST1],
     ([KEY=KEY1],
        [KEY ELEMENT and ALIST1->ALIST];
      enter(ALIST1,KEY,ELEMENT),
        [KEY1 ELEMENT1 and ALIST1->ALIST]);
  [KEY ELEMENT and ALIST->ALIST].
```

The operation to search in a list for the element with a given key should

1. return the element wanted, in case it is present in the list
2. fail, in case it is not present.

This can be expressed much better in CDL3 than in more conventional algorithmic languages:

```
TEST is in list(>ALIST,>KEY,ELEMENT>):
  [ALIST->KEY1 ELEMENT and ALIST1],
    ([KEY=KEY1];
     is in list(ALIST1,KEY,ELEMENT)).
```

We also add an operation to delete the element with a given key (if present):

```
FUNCTION delete from list(>ALIST>,>KEY):
  [ALIST->KEY1 ELEMENT1 and ALIST],
    ([KEY=KEY1];
     delete from list(ALIST,KEY),
       [KEY1 ELEMENT1 and ALIST->ALIST]);
    +.
```

Notice the use of an empty alternative, indicated by `+`.

Exercises

1. Improve this rather rudimentary implementation by realising the set as an ordered list.

2. improve it still more, by realising the set as a search tree (an ordered binary tree).

Chapter 3

Modules

In this chapter we describe the structure-in-the-large (see [DRE75]) of CDL3. Due to the presence of modules with interfaces, the language is eminently suitable for the **Bottom-UP programming** style, involving the programming of **abstract datatypes** (ADT's), as well as modular programming.

In this chapter, we will focus on the distinction between concrete and abstract datatypes, and the implementation of ADT's in CDL3. We will introduce a further concrete datatype, the **array**, which allows the efficient implementation of ADT's and give a number of typical examples.

3.1 Abstract and concrete datatypes

By a **datatype** (or *type*, for short) we mean a collection of values together with a number of operations on them.

Large data structures, of which in general only one instance exists (such as the symbol table in a compiler) are made visible only as a set of access algorithms. Objects of which more than one instance exists are either small (in the sense that they can conveniently fit into one word) or they are components of a larger data structure and access to them may well be provided through a small object (e.g. a pointer into a table).

Take stacks, as an example: preferably, the number of stacks used in a program should be at most two (for obvious engineering reasons). If we are in a situation where we have to deal with more than two stacks, it is just as much work to implement n stacks as it is to implement three: we may build an administration suitable for dynamically creating and manipulating any desired number of stacks, or even a stack of stacks, involving quite some hidden memory management. Each individual stack we will access through a pointer (either to its top or to some control block). Again the interface to such a data structure involves only access algorithms and small objects, which will conveniently fit within an affix.

In the sequel we will give a number of examples to elucidate this philosophy of data structures and its consequences.

3.2 Abstract datatypes

3.2.1 domain

3.2.2 denotation

3.2.3 construction or building-operation

3.2.4 acces-operation

3.2.5 representation-independence

3.2.6 signature

3.3 Modules in CDL3

Uncritically introducing a notation for global variables on top of the grammatical formalism is unsatisfactory, both from a formal viewpoint and from Software Engineering considerations.

In CDL3 a **module** is considered as an abstract data type that has precisely one instance. Since there is only one instance of it, the module need never be passed explicitly as a parameter.

The signature of the module is determined by its **interface**. The interface of a module specifies all its exported procedures and types, as well as the modules which will be used by this module.

As an instance of an abstract datatype, a module can contain local state information, in the form of global variables.

3.3.1 global variables

In CDL3, **global variables** can only be introduced in separate modules.

A module consists of one structured object (a special affix expression comprizing all its global variables) and the definitions of procedures and types to handle this object. The global variables are invisible transient parameters to all the procedures exported by the module and recursively of the procedures that call those procedures. Their introduction again serves to get rid of stereotypical parameters, and gain some security in the bargain.

example

```
MODULE lexico = buffered LINE.
```

The module has one global variable, named `LINE`.

3.3.2 interfaces

A module defines some **exported** meta-rules and procedures whose left-hand sides are given in a **specification**-list.

example

```
DEFINES TEST ahead (>TEXT),
        PRED is (>TEXT),
        ACTION should be (>TEXT),
        IDF,
        PRED is identifier(IDF>),
        TEST is end of source.
```

The modules exports five procedures and a type. Notice that, since is identifier has a parameter of type `IDF` (a synonym for `INT`), we must also export that type, otherwise it would be impossible to call this rule.

3.3.3 import

To **import** the procedures and types defines by a module we simply write that we use that module.

example

Any module needing something from the module lexico will contain the line

```
USES lexico.
```

3.3.4 Preludes and postludes

A module may have special procedures, the **preludes**, **interludes** and **postludes** of the module, which will be called respectively at the begining, before the second pass (see 7.4.1) and at the end of the execution of the program.

The introduction of preludes and postludes allows to factor out the initializations and finalizations from the program. The programmer need not worry globally about propagating and calling initializations and finalizations, and can concentrate on the main algorithm (*separation of concerns*).

Notice that the global variables are implicit parameters (respectively output, transient, input) of the preludes, interludes and postludes.

3.3.5 root module

The **root** (main procedure) is contained in a **root module**. Normally this module begins by `ROOT` followed by the name of the root. This root must not fail and must not have parameters.

example

The root module of the interpreter (see 6.3.2) begins by

```
ROOT program.
```

The CDL3 compiler is called with the name of the root module as a parameter. By means of the USES section, it finds which modules must be compiled together with the root module.

3.4 Example: lexical module

As a simple example of a module, just to show the notation used, we will provide a set of lexical analysis procedures for parsing-like purposes.

```
MODULE lexico = buffered LINE.

DEFINES TEST ahead (>TEXT),
        PRED is (>TEXT),
        ACTION should be (>TEXT),
        IDF,
        PRED is identifier(IDF>),
        TEST is end of source.
```

No other modules are used by this one, so there is no USES-part.

```
LINE, IDF = TEXT.

PRELUDE read buffer:
  read line(LINE), skip layout;
  [ "" -> LINE ].
```

The predefined `PREDICATE read line(TEXT>)` attempts to read one line of text from the input file (usually standard in) and will fail at the end of the file. The line will end on a new-line character, according to UNIX conventions. Therefore `LINE` can only obtain an empty value through an explicit assignation, used here to indicate that an end-of-file has been found.

```
TEST is end of source:
  [LINE -> ""] .

TEST ahead (>TEXT):
  is prefix(TEXT, LINE, LINE1).

PRED is (>TEXT):
  is prefix(TEXT, LINE, LINE), skip layout.
```

The `TEST is prefix (> TEXT1, >TEXT, TEXT2>)` is a predefined operation, which succeeds if the value of `TEXT1` is a prefix of `TEXT`, and yields the rest of `TEXT` as `TEXT2`; a conditional **dis-concatenation**.

```
ACTION should be (>TEXT):
  is (TEXT);
  write(TEXT + " expected").
```

The concatenated texts are written to standard out.

```
ACTION skip layout:
  is (" ");
  is prefix("\n", LINE, LINE), read buffer;
  +.

L, D = TEXT.
```

Some more synonyms for `TEXT`.

```
PRED is identifier(L+IDF>):
  is letter(L), identifier tail(IDF).

ACTION identifier tail(IDF>):
  is letter (L), identifier tail(IDF), [ L+IDF ->IDF ];
  is digit (D), identifier tail(IDF), [ D+IDF ->IDF ];
  skip layout,  [ "" -> IDF ].

PRED is letter(L>):
    prefix(LINE, 1, L, LINE1), between(L, "a", "z"),
      [LINE1 -> LINE].
```

The predefined function `TEST prefix(>TEXT, >INT, TEXT1>, TEXT2>)` puts the first `INT` characters of `TEXT` into `TEXT1` and the rest into `TEXT2`. The `TEST between (>TEXT, >TEXT1, >TEXT2)` checks whether the value of `TEXT` lies between that of the other two (inclusive).

```
PRED is digit(D>):
    prefix(LINE, 1, D, LINE1), between(D, "0", "9"),
      [LINE1 -> LINE].
```

3.5 The ARRAY constructor

In CDL3, **arrayss** are introduced as a **changeable mapping** from integers to affixes. This provides a convenient mechanism for the construction of datastructures, in which the integers play the role of **pointers**.

3.5.1 Array declaration

A global variable can be declared as an **array** i.e. a collection of affixes of the same type that are accessed via an integer index. This feature is allowed only for global variables, i.e., it cannot be used for free variables or parameters. To declare an array, just put {} after its name in the global variables list.

Example

```
MODULE graph = NODE{ } ACTIVE{ }.
DEFINES NODE, ARC, NODENR,
        blabla.

NAME = TEXT.
NODE :: node NAME ARCS.
ARCS :: ARC ARCS; empty.
ARC :: arc NAME to NODENR.
NODENR :: INT.
ACTIVE:: yes;no.
```

3.5.2 Length of an array

An array has at each moment a number of elements, its length, which is initially zero. Each element is numbered by an index >= 1 and <= the length of the array, and is of the parameter type of the array.

An array is automatically extended at the high-index end whenever we initialise an element that did not exist before (whose index is > the length of the array).

There is no a priori limit on the number of elements of an array (implementation as a logarithmic list).

3.5.3 access

The element with a given index INT can be accessed by

```
ARR{INT}
```

.

3.6 Examples

3.6.1 A binary tree

The nodes of a binary tree are put in an array: the successors of the node at position NR are put in positions $2 * NR$ and $2 * NR + 1$.

```
MODULE tree=NODE { }.

DEFINES KEY,
        ACTION enter (>KEY),
        TEST find (>KEY).

KEY     =  TEXT.
NR      =  INT.
NODE    :: empty;
           KEY.
```

Notice that the type NODE is not exported.

```
PRELUDE empty tree:
  [ empty->NODE{1} ].

ACTION enter (>KEY):
  enter (1, KEY).

ACTION enter (>NR, >KEY):
  [ NODE{NR}->KEY1 ],
    ([ KEY=KEY1 ];
     before (KEY, KEY1),
       enter (NR*2, KEY);
     enter (NR*2+1, KEY));
  [ NODE{NR}->empty ],
```

```
    [ KEY->NODE{NR} ],
    [ empty->NODE{2*NR} ],
    [ empty->NODE{2*NR+1} ];
  ?.
```

Notice that the children of a node at the front of the tree obtain welldefined (empty) values. The algorithm is written such that it can never acces an uninitialized element of the array. The third alternative (abort) is therefore logically superfluous, but it is included as a safeguard against possible stupid modifications by others in the future.

```
TEST find (>KEY):
  find (1, KEY).

TEST find (>NR, >KEY):
  [ NODE{NR}->KEY1 ],
    ([ KEY=KEY1 ];
     before (KEY, KEY1),
       (find (NR*2, KEY);
        -);
     find (NR*2+1, KEY));
  [ NODE{NR}->empty ],
    -;
  ?.
```

A small driver to exercise our search tree:

```
ROOT index.
USES tree.

ACTION index:
  write (">"),
    read line (TEXT),
    do (TEXT),
    index;
  +.

ACTION do (>TEXT):
  is prefix ("+", TEXT, KEY),
    enter (KEY);
  find (TEXT),
    write ("Found "),
    write (TEXT);
  write ("Could not find "),
    write (TEXT).
```

Chapter 4

Syntax-Driven programming

In this chapter we will be concerned with the construction of parsing procedures in CDL3. In this way we will illustrate the notion of **syntax-driven programming**.

4.1 Context-Free Grammars and their notation

We start out from the well-known BNF notation for Context-Free grammars. A grammar for a fragment of the English language might, in this notation, look like

⟨sentence⟩ ::= ⟨subject⟩ ⟨verb⟩ ⟨object⟩
⟨subject⟩ ::= the man | the dog
⟨verb⟩ ::= bit
⟨object⟩ ::= the man | the dog | his wife

This grammar consists of four rules. The words enclosed between angled brackets are the **nonterminal symbols**, serving as names for concepts. Words occurring without such brackets are to be taken literally, they are called **terminal symbols**. The curious sign ::= separates the *left-hand side* from the *right-hand side* of a rule. A right-hand side consists of one or more **alternatives**, separated by a vertical bar. an alternative is a sequence of **members**. Productions of one same left-hand side can be taken together, separating the *alternative* right-hand sides by a vertical bar.

Given a BNF grammar for some language, we can **generate** sentences of the language, check whether a given string is a sentence of that language, determine the structure of a sentence and depict its structure.

4.1.1 Generating

Following this grammar, we can generate sentences like: "the man bit the man" or "the dog bit the man".

Starting from the nonterminal symbol ⟨sentence⟩, we repeatedly substitute for some nonterminal symbol one of its alternatives (doing one **derivation step**), e.g.:

⟨sentence⟩
→ ⟨subject⟩ ⟨verb⟩ ⟨object⟩
→ the man ⟨verb⟩ ⟨object⟩
→ the man bit ⟨object⟩
→ the man bit the man

This generation process stops when there is no further nonterminal symbol to be substituted for. We have then derived from the initial symbol ⟨sentence⟩ by repeated derivation steps according to the grammar a sequence of (zero or more) terminal symbols, a **sentence**.

At any point in the generation process we have a string of terminal and nonterminal symbols (a **sentential form**) from which we may choose any nonterminal for the next derivation step. The order in which derivation steps are performed is immaterial. In the example we have always chosen the leftmost nonterminal (it is a **leftmost derivation**).

4.1.2 Drawing syntactic structures

A pictorial representation of the derivation of a sentence, in the form of a tree whose root is the initial symbol, whose nodes are marked with nonterminal symbols and whose arcs represent the derivation applied, is called a **syntax tree** or parse tree. A simple example is shown in figure 4.1.

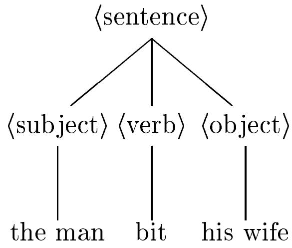


Figure 4.1: A syntax tree

Notice that the syntax tree abstracts away from the derivation order. All possible orders lead to the same syntax tree.

4.1.3 Analysis

The reverse of the derivation process is called analysis. We can try to **recognize** whether a given sequence of symbols is a sentence. Furthermore, if it is a sentence, we can **parse** it to build a syntax tree, according to the rules of the grammar. Obviously, recognition is a less ambitious process than parsing, since it gives only a yes/no answer and does not attempt to build up a syntax tree. In the next section we will introduce a simple and well-known parsing technique.

4.2 Recursive Descent parsing

Assume the following grammar, consisting of only one rule:

⟨sentence⟩ ::= a ⟨sentence⟩ | b ⟨sentence⟩ | #

This grammar generates the infinite language

$$L = \{\#, a\#, b\#, ab\#, ba\#, aa\#, \ldots\}$$

A naive recognizer for this language can be obtained by writing a CDL3 predicate:

```
PRED is sentence:
    is letter a symbol, is sentence;
    is letter b symbol, is sentence;
    is hash symbol.
```

This can be seen as a procedure yielding a Boolean value, which calls upon three other procedures (`is letter a symbol`, `is letter b symbol`, `is hash symbol`) which each recognize the symbol that their name suggests: `is letter a symbol` should have a realization like

if *next character of input* = "*a*"
  then *increment input pointer and yield true*
  else *yield false*

An input sentence like `ab#` would be recognized in the following fashion (in this trace the `>` sign before a name denotes calling the procedure of that name; the `+` sign denotes returning successfully, the `-` sign denotes returning with false).

```
> is sentence
 > is letter a symbol
 + is letter a symbol        a
 > is sentence
  > is letter a symbol
  - is letter a symbol
  > is letter b symbol
  + is letter b symbol        b
  > is sentence
   > is letter a symbol
   - is letter a symbol
   > is letter b symbol
   - is letter b symbol
   > is hash symbol
   + is hash symbol           #
  + is sentence
 + is sentence
+ is sentence
```

This simple example might lead one to believe that any Context-Free grammar can be recognized by a (recursive) combination of recognizers. This, unfortunately, is not the case.

Let us follow a trace of calls for the input `abx#`

```
> sentence
 > is letter a symbol
 + is letter a symbol       a
 > is sentence
  > is letter a symbol
  - is letter a symbol
  > is letter b symbol
  + is letter b symbol      b
  > is sentence
   > is letter a symbol
   - is letter a symbol
   > is letter b symbol
   - is letter b symbol
   > is hash symbol
   - is hash symbol
```

```
    - is sentence
  - is sentence
 -  is sentence
```

The recognizer fails, as indeed it should, but `a` and `b` have been read. Why is this a problem?

4.2.1 Backtracking

Some terminology: A predicate `x` is a **recognizer** for some nonterminal if, in case the input starts with a terminal production of that nonterminal, the predicate will consume that production from the input and return true, whereas otherwise it returns false.

An **exact recognizer** is a recognizer that, upon returning false, leaves the input as it was.

Assume the following, even simpler grammar

$\langle x \rangle$ ::= a b | c

Obviously,

$$L(x) = \{ab, c\}$$

We assume that we possess exact recognizers for `a, b` and `c` and construct the corresponding recognizer

```
PRED x : a, b; c.
```

This naive recognizer will unfortunately recognize the language

$$L'(x) = \{ab, c, ac\}$$

which was obviously not our intention.

Two possible solutions offer themselves:

- The first is to use a different type of parser, which performs explicit **backtracking** to reset the input. It could work schematically as follows

  ```
  PRED x:
      a, ( b; backtrack over a, c ); c.
  ```

  Such a backtrack parser may however be extremely inefficient and, what is worse, for many grammars it may still allow unintended parsings. Better parsing strategies are available,but we will not pursue this possibility here.

- To introduce **error productions**.

  Upon looking more closely at the problem, we notice that once an `a` has been recognized, a `b` should always follow. If a `b` does not follow, we know the input is incorrect. We will report this fact to the outside world by adding an error production :

  ```
  PRED x :
      a, (b; report error); c .
  ```

We will therefore be constructing recognizers with three possible outcomes:

- If it returns true without giving an error message, it has successfully recognized a sentence.
- If it returns false, it has not succeeded in recognizing a sentence and we know for sure that there was no sentence to recognize.
- If it returns true after having given an error message, it has successfully recognized an incorrect input as *incorrect*, which is equivalent to the second case.

A more fundamental way of looking upon this solution is to notice that we have extended the language with the collection of all erroneous strings, and that all erroneous strings will be reported as such.

We actually have now a three-valued recognizer. Of course the second and third alternative can easily be combined to give a two-valued exact recognizer.

4.2.2 Constructing a recursive descent recognizer

Let us extend the previous grammar with error productions.

```
PRED is sentence:
    is letter a symbol,
        ( is sentence;
          error ("Incorrect sentence") );
    is letter b symbol,
        ( is sentence;
          error ("Incorrect sentence") );
    is hash symbol.
```

We will distinguish two styles of recognizers. The previous example is a recognition predicate, with its error productions it is rather complicated. Another style for expressing the same is in the form of **recognising predicates** (whose name will conventionally start with `is`) and **recognising actions** (whose name will start with `should be`).

```
ACTION should be sentence:
    is letter a symbol, should be sentence;
    is letter b symbol, should be sentence;
    is hash symbol;
    error ("Incorrect sentence").
```

The second procedure is a recognizer for the language we intended. For incorrect input it gives an error message (upon which the state of the input is immaterial).

The recognition procedures obtained in this way form a **recursive descent parser**, which will work correctly for all CF grammars satisfying the so-called **LL(1) restriction** [ASU86], which can be summarized as follows:

1. Each alternative of a rule should have a different one-symbol look-ahead (so that the choice between alternatives can always be made on the basis of the next symbol of input).
2. Only one of the alternatives of a rule may have an empty terminal production (since this always succeeds) and this alternative should come last.

Given a CF grammar satisfying the LL(1) property, it can be turned into a recursive descent recognizer by doing the following **recognizer construction**:

- specify the root of the grammar as `ACTION`
- specify each rule having a nonfailing alternative (producing empty or calling only actions) as `ACTION`
- specify the remaining rules as `PREDICATE`
- put the resulting CDL3 program through the translator
- make a suitable error production for each defect reported

until all defects have been eliminated.

This process is so simple that no further compiler generation tool is necessary to construct recursive descent parsers for LL(1) grammars in CDL3.

4.2.3 When is an error production necessary?

For the sake of the discussion, assume a production rule

$\langle x \rangle ::= \ldots \langle p \rangle \ldots$

with its corresponding recognizer

```
PRED is x : ... , is p, ... .
```

In this alternative `is p` is a predicate. Obviously, before this predicate we can allow only tests and functions, since they have no side effects. After this predicate, we cannot allow the alternative to fail, so we can allow only actions or functions. At any rate, there may never be two predicates in one alternative, because that is sure to lead to a backtrack situation.

Let us give the name **activity** to any change in the observable environment. Of course what constitutes the observable environment (global variables, files or output media) is a matter of arbitrary human decision. In the context of parsing, the activity we are mainly interested in is the consumption of the input.

We will use the term **effect** for an activity upon success and the term **defect** for an activity upon failure. The backtrack philosophy of CDL3 is very simple: all defects are forbidden!

There is, as it were, a positive trace along which all intended computations lie. There should be no activities other than on this positive trace through the computation.

4.3 Sketch of a small interpreter

As an exercise we will sketch the structure of a small interpreter. In this process we will introduce other CDL3 constructs, so that the original syntactic motivation of the language becomes clear.

The interpreter will serve to recognize and execute straight-line programs in a toy language with the following syntax:

⟨program⟩ ::= ⟨statement sequence⟩ **.**
⟨statement sequence⟩ ::= ⟨statement⟩ **;** ⟨statement sequence⟩ | ⟨statement⟩
⟨statement⟩ ::= ⟨assignation⟩ | ⟨write command⟩
⟨assignation⟩ ::= ⟨variable⟩ = ⟨expression⟩
⟨variable⟩ ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
n | o | p | q | r | s | t | u | v | w | x | y | z
⟨write command⟩ ::= **!** ⟨variable⟩

The syntax of ⟨expression⟩ will be dealt with later (in 4.3.3). The semantics of this toy language will be described as we develop this example.

In order to construct an interpreter for the language, we start out by constructing a recognizer for ⟨program⟩s.

4.3.1 Constructing the recognizer

We obtain the recognizer by applying the construction algorithm given in 4.2.2 to the above grammar:

```
ACTION should be program:
    should be statement sequence, should be end symbol.

ACTION should be statement sequence:
    is statement sequence;
    error ("Incorrect statement"), ?.
```

Notice that we do not give a message “Incorrect statement sequence”; although this would be technically correct it is psychologically preferable to report a “smaller” error.

We have to apply **left factorization** to the rule for ⟨is statement sequence⟩, taking out the common left-factor ⟨is statement⟩:

```
PRED is statement sequence:
    is statement,
       (is semicolon symbol, should be statement sequence;
       +).

PRED is statement:
    is assignation;
    is write command.

PRED is assignation:
    is variable, should be equals symbol,
       should be expression.

PRED is write command:
    is write symbol, should be variable.
```

In a next step we will turn this recognizer into the **driver** of the interpreter by extending it with some actions which perform the desired interpretation as a side effect of the recognition process. But first we make a digression about the appropriate reading strategy for the input.

4.3.2 Reading ahead

We will have to implement a number of procedures which read and recognize part of the input. At each position of the input different symbols may stand, indicating different syntactic constructions. The recognition routines will have to read those symbols in order to look at them.

It is not a good idea to let each recognition procedure read a symbol, only to find out in many cases that it cannot use this symbol. We will make use of a module (to be defined later) which encapsulates a systematic read-ahead method, using a buffer which at any point in time contains the first nonconsumed symbol of the input. Whenever a symbol is recognized, the next symbol of the input is read into this variable. This look-ahead buffer will be provided with the following interface:

- `TEST ahead (>SYMB)`
  succeeds iff the buffer contains the symbol `SYMB`
- `PRED is (>SYMB)`
  like `ahead`, but upon success also reads a new symbol from input and stores this in the buffer
- `ACTION should be (>SYMB)`
  like `is`, but gives an error message in case the look-ahead fails.

Using these procedures, the recognizers for the lexical symbols in the example (in 6.2) can be written: the procedures

```
PRED is semicolon symbol:
    is (";").

ACTION should be equals symbol:
    should be ("=").
```

and similarly for the others.

4.3.3 Syntax of expressions

The interpreter deals with expressions, that have to be read and whose value has to be computed. We will assume the following syntax for expressions, similar to the one in PASCAL but somewhat simplified.

$\langle\text{expression}\rangle ::= \langle\text{expression}\rangle\ \langle\text{plusminus}\rangle\ \langle\text{term}\rangle \mid \langle\text{term}\rangle$
$\langle\text{plusminus}\rangle ::= + \mid -$
$\langle\text{term}\rangle ::= \langle\text{term}\rangle\ \langle\text{timesby}\rangle\ \langle\text{factor}\rangle \mid \langle\text{factor}\rangle$
$\langle\text{timesby}\rangle ::= \times \mid \div$
$\langle\text{factor}\rangle ::= \langle\text{variable}\rangle \mid \langle\text{constant}\rangle \mid \mathbf{(}\ \langle\text{expression}\rangle\ \mathbf{)}$

The straightforward transliteration of the syntax rule for ⟨expression⟩ into a recognition predicate leads to

```
PRED is expression:
    is expression, is plusminus, is term;
    is term.
```

We immediately notice two places which give rise to **defects**, which we can solve by using actions rather than predicates.

But there is a more serious flaw. Due to the **left-recursion**, this procedure will loop endlessly! In fact the grammar in this form does not satisfy the LL(1) restriction.

In order to overcome this problem, we shall now introduce a standard technique from the field of Compiler Construction for the elimination of left-recursion.

4.3.3.1 Left-recursion elimination

Consider the left-recursive syntax rule

⟨s⟩ ::= ⟨s⟩ a | b

and the language generated by it:

$$L(s) = \{b, ba, baa, baaa, \ldots\}$$

There is an obvious regularity to this language, which tells us that we can generate the same language by means of the two rules

⟨s⟩ ::= b ⟨s'⟩
⟨s'⟩ ::= a ⟨s'⟩ |

which are no longer left-recursive. The second grammar is **weakly equivalent** to the first one, in the sense that it describes the same language but does not necessarily assign the same syntax trees. Every left recursion can be eliminated from a CF grammar by this transformation, albeit in some cases at the price of a serious increase in the size of the grammar.

Let's now attempt to remove the left-recursion from the syntax rule for ⟨expression⟩.

4.3.3.2 Application to expressions

Applying the previous transformation, we can rewrite the rule for ⟨expression⟩ to two rules

⟨expression⟩ ::= ⟨term⟩ ⟨expression tail⟩
⟨expression tail⟩ ::= ⟨plusminus⟩ ⟨term⟩ ⟨expression tail⟩ |

which together describe the same language. The rule for ⟨expression tail⟩ is right-recursive, but this causes no problem in recognition. A similar transformation is to be applied to ⟨term⟩, introducing a ⟨term tail⟩. The resulting grammar is now indeed of type LL(1).

4.3.3.3 Why is expression left-recursive?

A naive person might find attempt the following way to replace the left-recursion in ⟨expression⟩ by right-recursion: Consider the syntax rule

⟨expression⟩ ::= ⟨expression⟩ ⟨plusminus⟩ ⟨term⟩ | ⟨term⟩

Exchange two members and we have right-recursion

⟨expression⟩ ::= ⟨term⟩ ⟨plusminus⟩ ⟨expression⟩ | ⟨term⟩

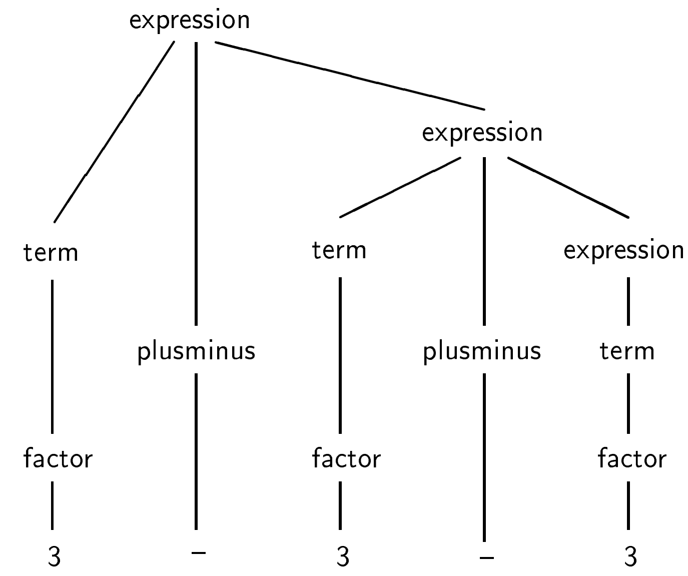


Figure 4.2: right-recursive syntax tree

and the language is precisely the same!

The language may be the same but the syntax trees are definitely not the same. And if the semantics follows the syntactic structure, in the sense that the value of the expression is recursively expressed in terms of the values of its parts (the principle of **compositionality**), then the two interpretations may lead to different results.

Consider as an example the expression $3-3-3$. According to the original left-recursive syntax, its value is that of $(3-3)-3$, i.e. minus three; according to the naive right-recursive syntax it is $3-(3-3)$, i.e. plus three! The first interpretation is the intended one.

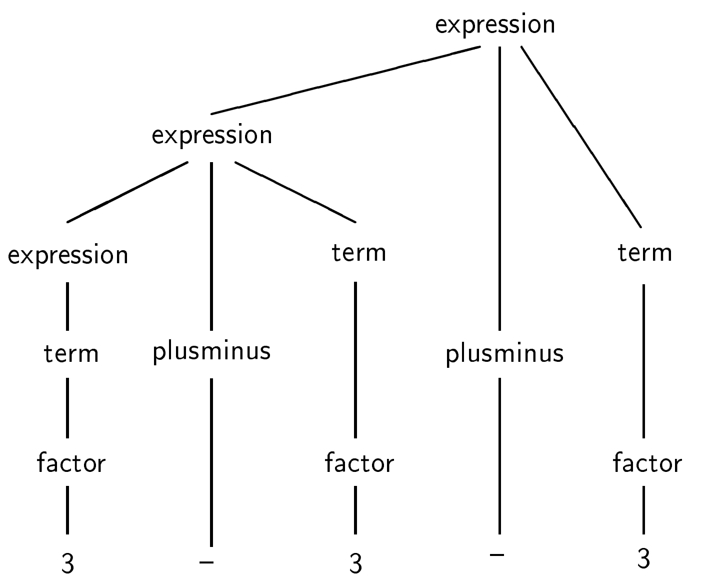


Figure 4.3: left-recursive syntax tree

It appears that we must demand from our left-recursion elimination that it preserves compositionality. We will presently show for the non-naive left-recursion elimination rule that this can be achieved.

4.3.4 Recognizer for expression

The transliteration of the first rule of the transformed grammar into a recognition predicate leads to

```
PRED is expression:
    is term, is expression tail.
```

This rule contains a defect, which can best be solved by attempting to turn it (and most of the other rules) into **recognition actions** rather than predicates. Each action will be invoked in a situation where it should succeed if the input is correct. Furthermore, there are optional parts which *may be* present. The task of protesting against incorrect input is delegated as far down as possible.

```
ACTION should be expression:
    should be term, may be expression tail.

ACTION may be expression tail:
    is plusminus, should be term, may be expression tail; +.
```

The second rule may (by **folding** the last two members of the first alternative) be simplified into

```
ACTION may be expression tail:
    is plusminus, should be expression; +.
```

The transliteration of the remaining syntax rules is as follows:

```
PRED is plusminus:
    is ("+"); is ("-").

ACTION should be term:
    should be factor, may be term tail.

ACTION may be term tail:
       is times by, should be term; +.

PRED is timesby:
    is ("*"); is ("/").

ACTION should be factor:
    is variable;
    is constant;
    is ("("), should be expression, should be (")");
    error ("Incorrect expression").
```

Notice that only one explicit error production is introduced.

We have now obtained a recursive descent recognizer for expressions from the original CF grammar by systematically applying rules for turning each rule into a recognition procedure, giving it a suitable type and eliminating defects by left-factorization and the introduction of error alternatives.

Chapter 5

Example: A text editor

As an example combining syntax-driven programming with modular design, we shall develop a simple text editor. We choose this particular example because an editor is a typical interpreter, with a user interface which can be implemented as a collection of semantic actions (performing table administration, I/O etc.) controlled by the syntactic structure of the input.

5.1 The user interface

The task is to design and implement a very simple editor. The editor is line oriented, and deals with a *text* consisting of *lines*, each line containing a *number* and an *entry*. Lines should be kept in the order of increasing line numbers.

The editor has four types of commands:

- The insert-command <numb>=<entry><RET> where

  <numb> is some positive integer (the line number),

  = serves as a separator, and

  <entry> is some sequence of characters, ending on

  <RET> the end-of-line character.

  Effect: If a line with line number <numb> is present in the text, it is overwritten with this <entry>. Otherwise, a line with number numb and this entry is inserted.

- the delete-command d<numb><RET> where

  d is the letter d, and

  <numb> is a line number.

  Effect: The line with number <numb> is deleted from the text if present, otherwise an error is reported.

- the list-command l<RET> where l is the letter l.

  Effect: All lines of the text are listed in the order of increasing line number.

- the quit-command q<RET> where q is the letter q.

  Effect: The editor halts, nothing is saved.

This editor is so crude that nobody would like to work with it, but in its design some important issues are addressed which, due to its simplicity, are not smothered by irrelevant complexity and detail.

5.2 Top-Down exploration

We shall first make a Top-Down exploration to find the elements of the problem. At the highest level of abstraction, the editor is a simple interpreter which reads one command line at a time and executes it, until it finds the quit-command. We have to take into account something which is customarily left out in the problem specification, viz. the fact that human users of an interface make errors. We therefore have to include error treatment and deal with such issues as the skipping of blanks.

```
ACTION edit:
  skip blanks,
    (is insert command, edit;
     is delete command, edit;
     is list command, edit;
     is quit command;
     incorrect command, edit).
```

Notice that we have introduced one error alternative.

The above algorithmic structure is somewhat harder to achieve in languages with conventional control structures than inCDL3. In case the reader feels put off by the many right-recursions in this procedure, she may be happy to learn that all right-recursions will be resolved by the implementation ofCDL3.

In realizing the various commands, we shall try to separate the recognition of a command from its execution, the two phases communicating through affixes. This principle is clearly shown in the following procedure.

```
PRED is insert command:
  is number(NUMBER),
    skip blanks,
    should be equals sign,
    get line(TEXT),
    store(NUMBER,TEXT).
```

In the true Top-Down style, for as far as we can not make use of standard operations, as we go along we shall postulate operations to be defined later.

```
PRED is delete command:
  is delete symbol,
    skip blanks,
    (is number(NUMBER),
       delete(NUMBER),
         skip rest of line;
     incorrect deletion command).

ACTION incorrect deletion command:
  screen("Line number expected\n"),
    skip rest of line.
```

The action `skip rest of line` is used here to deal with the possibility that, after the deletion command but before the end of line, some rubbish will have to be skipped. We might have chosen another convention instead, like allowing more than one command per line.

```
PRED is list command:
  is list symbol,
    list,
      skip rest of line.

PRED is quit command:
  is quit symbol.

ACTION incorrect command:
  screen("(Q)uit,(D)elete,(L)ist or insert command expected\n"),
    skip rest of line.
```

5.3 Subdivision into modules

We have been inventing new elementary actions and predicates as we went along. We shall now classify these as belonging to two different modules: a module `symbol recognition` to deal with the recognition of symbols and a module `data management` to deal with the storage and retrieval of data.

The abstract operations for input/output are:

```
ACTION skip blanks
ACTION get line(TEXT>)
PRED is delete symbol
PRED is list symbol
PRED is quit symbol
ACTION should be equals sign
PRED is number(INT>)
ACTION skip rest of line
```

Those dealing with data management are:

```
ACTION store(>NUMBER,>TEXT),
TEST get(>NUMBER,TEXT>),
ACTION delete(>NUMBER),
ACTION list.
```

We shall define all those operations in appropriate modules. The program as it stands can be turned into a module in its own right, by prefixing it with

```
MODULE editor.
DEFINES ACTION edit.
USES transput,
     data management.
```

5.4 Symbol recognition

The module `symbol recognition` will serve as an interface to the standard**transput** (= input/output) (see appendix B) which supplies capabilities for line-oriented file transput, as well as interactive character-oriented transput from the keyboard and to the screen.

Based on those capabilities, the module will have to provide the recognition- and reading operations introduced above.

```
MODULE symbolrecognition.
DEFINES ACTION skip blanks,
        ACTION get line(TEXT>),
        PRED is delete symbol,
        PRED is list symbol,
        PRED is quit symbol,
        ACTION should be equals sign,
        PRED is number(INT>),
        ACTION skip rest of line.
USES inout.

ACTION get line(TEXT>):
  is symbol("\n"),
    [""->TEXT];
  is end of file,
    [""->TEXT];
  is any char(TEXT),
    get line(TEXT1),
    [ TEXT+TEXT1->TEXT ].

PRED is delete symbol:
  is symbol("d").

PRED is list symbol:
  is symbol("l").

PRED is quit symbol:
  is symbol("q").

PRED is number(INT>):
  ahead("0","9"),
    rest number, get(TEXT),
      (decbin(TEXT,INT);
       [0->INT]).

ACTION rest number:
  ahead("0","9"),
    rest number;
  +.

ACTION skip blanks:
  is symbol(" "),
    skip blanks;
```

```
  is symbol("\t"),
    skip blanks;
  +.

ACTION should be equals sign:
  is symbol("=").

ACTION skip rest of line:
  is end of file;
  is symbol("\n");
  is any char(TEXT),
    skip rest of line.
```

5.5 Data management

The module `data management` serves to supply the storage and retrieval of numbered lines, as well as their printing. For the latter purpose it is also based on `inout`.

```
MODULE datamanagement=TABLE.
DEFINES NUMBER,
        ACTION store(>NUMBER,>TEXT),
        TEST get(>NUMBER,TEXT>),
        ACTION delete(>NUMBER),
        ACTION list.
USES inout.
```

The central datastructure, which is a global variable of the module, is an ordered list of `<NUMBER, TEXT>` pairs.

```
TABLE :: empty;
         NUMBER TEXT and TABLE.
NUMBER:: INT.

PRELUDE clear table:
  [empty->TABLE].

ACTION store(>NUMBER,>TEXT):
  store(TABLE,NUMBER,TEXT).

TEST get(>NUMBER,TEXT>):
  get(TABLE,NUMBER,TEXT).

ACTION delete(>NUMBER):
  delete(TABLE,NUMBER).

ACTION store(>TABLE1>,>NUMBER,>TEXT):
  [TABLE1->empty],
    [NUMBER TEXT and TABLE1->TABLE1];
  [TABLE1->NUMBER1 TEXT1 and TABLE2],
    ([NUMBER=NUMBER1],
       [NUMBER1 TEXT and TABLE2->TABLE1];
```

```
     less(NUMBER,NUMBER1),
       store(TABLE2,NUMBER,TEXT),
       [NUMBER1 TEXT1 and TABLE2->TABLE1];
     [NUMBER TEXT and TABLE1->TABLE1]).

TEST get(>TABLE1,>NUMBER,TEXT>):
  [TABLE1->NUMBER1 TEXT and TABLE2],
    ([NUMBER=NUMBER1];
     less(NUMBER,NUMBER1),
       get(TABLE2,NUMBER,TEXT)).
```

Deleting the element with a given key has no effect if that key was already present.

```
ACTION delete(>TABLE1>,>NUMBER):
  [TABLE1->NUMBER1 TEXT and TABLE2],
    ([NUMBER=NUMBER1],
       [TABLE2->TABLE1];
     less(NUMBER,NUMBER1),
       delete(TABLE2,NUMBER),
       [NUMBER1 TEXT and TABLE2->TABLE1];
     screen(I cannot delete the line number "),
       screen int(NUMBER),
       screen("\n")).

ACTION list:
  list(TABLE).

ACTION list(>TABLE1):
  [TABLE1->NUMBER TEXT and TABLE2],
    list(TABLE2),
    screen int(NUMBER),
    screen(":",TEXT,"\n");
  +.
```

5.6 Conclusion

The intention of this example was to show the design strategy leading to a structure of modules, each with their specific tasks. A more complete and realistic example is given in Appendix C.3.

Chapter 6

Transducer construction

By itself a recognizer is a very uninteresting algorithm, and rather useless because it only (at best) answers with yes or no. While our computer is going through the tremendous effort of recognizing expressions, it might just as well do something useful: perform some computation or produce some output.

The program can then be seen as an **interpreter**, a special case of a **transducer** as it is called in formal language theory [ASU86].

6.1 Transducers

Transducers are a formalization of string-to-string mapping expressed in the form of grammars. They can be described and classified in numerous ways.

6.1.1 Simultaneous Grammars

The simplest form of transducer is what we shall call a **simultaneous grammar**: CF grammar with two alphabets V_{in} and V_{out} of terminal symbols, called the input- and output vocabulary.

The grammar is an ordinary CF grammar over $V_t = V_{in} \bigcup V_{out}$ and as such describes a set string. Each such string consists of (zero or more) symbols of either alphabet; reading only the input-symbols one obtains an input-string; reading only the output-symbols gives the corresponding output string.

As an example we take a variant of the **Fibonacci grammar** describing all possible ways a string of n a's can be split up into pairs or single a's. We use the convention of underlining an input terminal and overlining an output terminal.

S: $\underline{\texttt{a}}$, S;
   $\underline{\texttt{a}}$,$\underline{\texttt{a}}$, S;
   .

Writing $\overline{s}$ for a single a and $\overline{p}$ for a pair, the following nondeterministic simultaneous grammar

S: $\underline{a}$, $\overline{s}$, S;
   $\underline{a}$, $\underline{a}$, $\overline{p}$, S;
   .

associates with the input aaaa the strings

ssss ssp sps pss pp.

A simultaneous grammar can only transducer sequences of input symbols to sequences of output symbols in a combinatorial fashion. For more general transductions we need more freedom, especially in the ordering of translated constituents.

6.1.2 CF transducers

A **CF transducer** consists of a Context Free grammar in which each alternative *alt* is equipped with a (possible empty) **transduction part** $trans_{alt}$.

The following conventions shall be used:

- In principle we indicate, next to each alternative and separated from it by a slash /, its transduction part.
- In the transduction part all the nonterminals of the alternative may be used (in any order), as well as (output) terminal symbols.
- the **translation** of a nonterminal consists of the translations of the members of its transduction part, in that order.
- A nonterminal in the transduction part denotes the translation of the same nonterminal in the alternative. In case of multiple occurrences of a nonterminal in an alternative, the various occurrences may be distinguished by a number as suffix.
- An output terminal symbol translates to that symbol.
- Input terminal symbols have no translation (unless a similar output terminal is explicitly given in the transduction part).
- If the translation of an alternative consists merely of the translations of its members, in that order, we omit the transduction part (to save writing).

Some examples:

1. In the CF transducer formalism the Fibonacci transducer can be expressed as

   ```
   S: a, S / "s" S ;
      a, a, S / "p" S ;
      .
   ```

2. As a more complicated example we give a grammar describing a well-known proof-reading convention: the use of **text inversions** like $|\overline{abc}|\underline{def}|$, which should be corrected to *def abc*.

   The input grammar describes strings of characters in which subsequences of the form `< ... | ... >` may appear; in transducing such an inversion, the segments to the left and to the right of the | are to be interchanged.

   ```
   sentence:
      character, sentence;
      inversion, sentence;
      .
   ```

```
character:
   "a" / "a"; "b" / "b" \ldots .

inversion: "<" text1 "|" text2 ">" / text2 text1.
```

3. The production of a parse tree can be seen as a canonical transduction to labeled brackets:

```
sentence:
   subject, verb, object /
          "sentence: (" subject "," verb "," object ")".
```

4. As a further example we shall sketch how to generate, instead of a parse tree, a semantic representation in the form of **frames**. In a highly simplified grammar for English:

```
descriptive sentence:
   subject, copula, predicate / predicate" ("subject")\n";
   subject, verb, {object} /
     "actor(" subject") action(" verb ") victim(" object ")\n".
```

Notice that the copula obtains no translation.

6.1.3 Invertible transductions

Under certain restrictions (the most important being that each nonterminal in the left-hand side appears precisely once in the right-hand side) we obtain an **invertible transducer**: by interchanging left- and right-hand side the inverse transduction is obtained. A similar property holds for pure PROLOG.

The inverse transduction may be deterministic where the original transduction is not (as in the Fibonacci transducer) or nondeterministic where the original is not (in case different inputs transduce to the same output). It is very useful for a transducer to be invertible, because it can the be used for translation in the other direction and for paraphrasing output sentences (e.g. semantical frames). The GWB will check whether a given transducer is invertible.

6.2 Interpreting expressions

Keeping in mind that we are building an interpreter, we will try to compute the value of the expression recognized as a side-effect of the recognition process.

We equip each of our recognizing predicates and actions with one output parameter, which will return the value computed. This value is going to be an integer.

```
V, VAL = INT.
```

In recognizing an expression, we shall first recognize a ⟨term⟩, obtaining a value which we shall pass to the ⟨expression tail⟩ in order to be able to compute the value of the ⟨expression⟩ in a compositional way.

```
ACTION should be expression (VAL>):
    should be term (VAL), may be expression tail (VAL).
```

The successfull recognition of a ⟨plusminus⟩ shall tell us which operator it was.

```
OP :: plus ; minus; times; div.

PRED is plusminus (OP>):
    is ("+"), [ plus -> OP ];
    is ("-"), [ minus -> OP ].
```

Between the members of the original syntax, we insert **semantic actions**, i.e. calls on actions that serve to operate on affix values during the recognition process.

```
ACTION may be expression tail (>VAL>):
    is plusminus (OP),
        should be term (V),
           compute (VAL, OP, V),
              may be expression tail (VAL);
    +.
```

The semantic action `compute (>VAL>, >OP, >V)`, which has to perform the actual computation, will be defined later. Observe that compositionality is preserved in this formulation.

```
ACTION should be term (VAL>):
    should be factor (VAL), may be term tail (VAL).

ACTION may be term tail (>VAL>):
    is timesby (OP),
       should be factor (V),
          compute (VAL, OP, V),
             may be term tail (VAL);
    +.

PRED is timesby (OP>):
    is ("*"), [ times -> OP ];
    is ("/"), [ div -> OP ].

ACTION should be factor (VAL>):
    is variable (VAL);
    is constant (VAL);
    is ("("), should be expression (VAL), should be (")");
    error ("Incorrect expression").
```

This last recognition action is not quite correct, because it must be such that it also yields a value in error situations: `VAL` may not obtain an undefined value. For lack of better we choose for zero.

```
ACTION should be factor (VAL>):
    is variable (VAL);
    is constant (VAL);
    is ("("), should be expression (VAL), should be (")");
    error ("Incorrect expression"), [ 0 -> VAL ].
```

6.3 Adding static semantics

In the field of compiler construction, the term **static semantics** is used for that part of the semantics of a programming language that is independent of the input but depends only on the program, and therefore can be dealt with at **compile time**. In particular, this holds for the identification of names, the attribution of types to the constructs of the program and the related consistency checks (**context conditions**).

While interpreting its input, our interpreter will perform computations over a certain semantical domain. We will first define an appropriate semantic domain and the operations over it, then add the calls of the semantic actions and percolate the necessary information to the places where it is used or modified.

6.3.1 Choice of semantic domain

For our toy language, the semantics will be very simple: there are 26 variables, each named by a small letter, which initially possess an undefined value. A variable can obtain an integer as value, by means of an assignation. The write-command prints the current value of the variable. The current value of variables will also be needed in the computation of the value of an expression.

We can realize this semantical domain as a *variable list*, a list of pairs <*name, value*>, containing only those names which have a defined value. In fact, this `VLIST` is just an association list as described in 2.6.

```
VLIST :: empty; NAME VAL and VLIST.
NAME  :: TEXT.
VAL   :: INT.
```

There will be one such `VLIST` passed in and out every recognition procedure in the driver. The operations on it shall be

`FUNCTION empty list (VLIST>)`	create an empty variable list
`FUNCTION take value (>VLIST, >NAME, VAL>)`	value lookup
`FUNCTION assign (>VLIST>, >NAME, >VAL)`	value modification.

We can obtain these by taking a modified copy of the operations defined in 2.6.

```
FUNCTION empty list (VLIST>):
  [empty -> VLIST].

FUNCTION assign (>VLIST>, >NAME, >VAL):
  [VLIST -> NAME1 VAL1 and VLIST1],
     ([NAME=NAME1],
        [NAME VAL and VLIST1 -> VLIST];
      assign (VLIST1, NAME, VAL),
        [NAME1 VAL1 and VLIST1 -> VLIST]);
  [NAME VAL and empty -> VLIST].

TEST take value (>VLIST, >NAME, VAL>):
   [VLIST -> NAME1 VAL and VLIST1],
      ([NAME=NAME1];
       take value (VLIST1, NAME, VAL));
   error ( "Variable " + NAME + " undefined"), ?.
```

6.3.2 Completing the interpreter

We will now modify the recognizer described in 4.3.1 so that during recognition it will call the necessary semantic actions to perform computations, keep up the variable list and print results. To that end, the interpreter should start out with an empty variable list, which is modified by assignations and consulted in certain places. This variable list must be percolated (as a parameter) to all places where it is needed.

```
ACTION should be program:
    empty list (VLIST),
       should be statement sequence (VLIST),
          should be end symbol.

PRED is statement sequence (>VLIST>):
    is statement (VLIST),
       (is semicolon symbol, should be statement sequence (VLIST);
       +).

ACTION should be statement sequence (>VLIST>):
    is statement sequence (VLIST);
    error ("incorrect statement"), ?.

PRED is statement (>VLIST>):
    is assignation (VLIST);
    is write command (VLIST).

PRED is assignation (>VLIST>):
    is variable (NAME),
       should be equals symbol,
         should be expression (VLIST, VAL),
            assign (VLIST, NAME, VAL).

PRED is write command (>VLIST):
    is write symbol, should be variable (NAME),
  take value (VLIST, NAME, VAL),
     print (VAL).
```

Similarly, the varable list must be percolated to the recognizer for expressions (notice that `should be expression` has obtained an extra parameter), to arrive at its use in:

```
ACTION should be factor (>VLIST, VAL>):
    is variable (VLIST, VAL);
    is constant (VAL);
    is ("("), should be expression (VLIST, VAL), should be (")");
    error ("Incorrect expression"), [ 0 -> VAL ].
```

We have to provide some further procedures, such as `is variable (NAME>)` but we leave their definition to the reader. Notice that the affix `VLIST` is passed in and out of practically each procedure. This makes it a good candidate for replacement by a global variable (see 3.3.1).

This completes our outline of the syntax-driven design and implementation of a simple interpreter.

6.4 Precedence directed parsing

If we have more operators, we may of course continue in the same style, adding one recognition procedure for each priority level. On the other hand it is better to exploit the priorities of operators to make a simple general recognition scheme.

We assume the operators to have the conventional priorities, as shown in figure 6.1.

priority	operators			
1	`:=` (left-associative!)			
2	`OR`			
3	`AND`			
4	$=$	$\neq$		
5	$<$	$>$	$\leq$	$\geq$
6	$+$	$-$		
7	$*$	$/$	`DIV`	
8	$**$			
9	monadic operators			

Figure 6.1: priorities of operators

In this priority system, a $\langle$term$\rangle$ has priority 6 and a $\langle$factor$\rangle$ priority 7.

Let us look at one operator in the middle of an expression, the $*$ in

$$a ** 2 - b * c ** 2 + d$$

How far do the operands of that multiplication extend? The operands are obviously those expressions immediately to the left and to the right of the operator whose operators have a priority *not lower* than that of the multiplication.

In a hybrid of mathematical and CDL notation we might write:

$$\mathsf{expression} : \mathsf{expr}_0.$$

$$\mathsf{expr}_i : \mathsf{expr}_{i+1}, (\mathsf{op}_i,\ \mathsf{expr}_i;\ ). \quad \text{for} \quad 0 \leq i < max - 1$$

$$\mathsf{expr}_{max} : \mathsf{primary}.$$

Notice that this formulation is non-deterministic (it is a specification rather than an algorithm). It is possible to eliminate the non-determinism and the left-recursion to obtain the following formulation in CDL3:

```
PRED is expression:
    is expr of priority (0).

P, PRIO = INT.
MAX = 10.

PRED is expr of priority (>PRIO):
    [ PRIO = MAX ],
     (  (is primary; -);
    is expr of priority (PRIO + 1),
        may be expr tail of priority (PRIO).
```

```
ACTION may be expr tail of priority (>PRIO):
    is operator (PRIO, OP),
       should be expr of priority (PRIO + 1),
          may be expr tail of priority (PRIO);
    + .

ACTION should be expr of priority (>PRIO):
    is expr of priority (PRIO);
    error ("expression expected").
```

Upon recognition of an operator, its priority has to be deduced:

```
PRED is operator (>PRIO, OP>):
    operator ahead (P, OP), [ P = PRIO ], next symbol.

TEST operator ahead (PRIO>, OP>):
    can be operator code (SYMB), [ SYMB -> OP ],
       get priority (OP, PRIO).
```

We have in this generalization captured all possible levels of priority, of course at the expense of somewhat more computing effort. The extra effort comes from the fact that we do not exploit the priority of the following operator directly, but try all possible higher priority operators.

A more intelligent and final version is the following

```
PRED is expr of priority (>PRIO):
    is primary,
        may be expression tail (PRIO).

NEXTP = PRIO.

ACTION may be expression tail (>PRIO):
   operator ahead (NEXTP, OP),
      ([ PRIO < NEXTP ], next symbol,
          should be expr of priority (NEXTP + 1),
             may be expression tail (PRIO);
       + );
   + .
```

This is also much more compact than the original formulation.

6.5 Translation

The next step beyond recognition (4.3.4) and interpretation (6.3.2) is translation and the related problem of syntax tree construction.

6.5.1 Translation of expressions

We will now look at the problem of translating expressions into the so-called **reverse Polish** form [ASU86].

We define the reverse Polish form of an expression as follows: Let e be an expression and $P(e)$ its reverse Polish form. Then $P(x)$ is defined by

- If x is a variable or a constant then $P(x) = x$.
- If x is $e_1 \odot e_2$ where e_1 and e_2 are subexpressions and $\odot$ is an operator, then $P(x) = P(e_1)\, P(e_2) \odot$.

As an example, the translation of

$$a + 5 * b - (4 + c) * d$$

into reverse Polish form is

$$a\ 5\ b\ *\ \ +\ 4\ c\ +\ d\ *\ -$$

Notice that in the reverse Polish form no brackets are needed. That is an important reason for introducing it. There is an obvious relationship between the reverse Polish form of an expression and a program for a stack computer: The operations `*`, `+` and `-` find both their operands on the stack and replace them by the result of the operation.

How then can we generate this reverse Polish form from the original infix expression?

The definition of the reverse Polish form shows that every simple element (variable, constant) is translated by itself. In the translation of a composed expression, the translations of the subexpressions are in the same order as the original subexpressions, while the operator must be put at the end of the translation.

Then we can output the simple elements as soon as they are read, while an operator must be kept in some variable and output after the translation of its second operand.

We start out from the recognizer in 4.3.4 and modify it so that it outputs the reverse Polish form as a side effect of the recognition process.

- Upon recognition of a factor, it is immediately output (produced).
- Upon recognition of an operator, nothing is output until both operands have been recognized (and produced) and only then is the operator produced.

```
ACTION should be expression :
    should be term,
       may be expression tail,

OP :: plus ; minus; times; div.

ACTION may be expression tail:
    plus minus (OP), should be expression, produce (OP);
    +.

ACTION should be term :
    should be factor,
        may be term tail.

ACTION may be term tail:
    is times by (OP), should be term, produce (OP);
    +.

VAR, CONST :: TEXT.
```

```
PRED should be factor :
    is variable (VAR), produce (VAR);
    is constant (CONST), produce (CONST);
    is open symbol, should be expression,
       should be close symbol.
```

By adding suitable actions `produce` we obtain a from infix expressions into reverse Polish form.

6.6 Generating an abstract syntax tree

We shall now describe how to construct a syntax tree, containing all the information about the structure of the analysed expression.

In fact, the tree produced need not be precisely the syntax tree according to the syntax of expression 4.3.3 because it need not contain a node for each rule used in the parsing process: some nonterminals (like term and factor) are not necessary or even useful for the understanding of the expression. Also some of the terminal symbols (the brackets) need not be present in the tree, since the information they convey is already expressed in the structure of the tree. We need a condensed version of the syntax tree, often called an **abstract syntax tree** (AST).

First we describe the structure of the abstract syntax tree, by means of one or more affix-rules.

```
EXPR :: var NAME ; const INT ; EXPR OP EXPR .
OP :: plus  ; minus ; mult ; div .
```

Next we construct a transducer from the (original) **concrete syntax** to an abstract syntax tree.

```
ACTION should be expression (EXPR>):
    should be term (EXPR), may be expression tail (EXPR).

ACTION may be expression tail (>EXPR>):
    is plusminus (OP),
        should be term (EXPR1),
           [ EXPR1 OP EXPR -> EXPR ]
              may be expression tail (EXPR);
    +.

ACTION should be term (EXPR>):
    should be factor (EXPR), may be term tail (EXPR).

ACTION may be term tail (>EXPR>):
    is times by (OP),
        should be factor (EXPR1),
           [ EXPR1 OP EXPR -> EXPR ]
              may be term tail (EXPR);
    +.

ACTION should be factor (EXPR>):
    is variable (NAME), [ var NAME -> EXPR];
    is constant (INT), [ const INT -> EXPR];
    is ("("), should be expression (EXPR), should be (")");
    error ("Incorrect expression").
```

Notice e.g. that two expressions that differ only by redundant parentheses have the same abstract syntax tree, as is proper because they have the same meaning.

6.7 Conclusion

In this chapter we have illustrated the notion of syntax-directed programming, touching upon some subjects from the field of compiler construction (parsing, semantic actions, transducers).

Our "methodology" for the syntax-directed construction of one-pass two-level transducers (**syntax-driven programming**) can be summarized as follows:

- construct the input recogniser
  - deduce a context-free grammar of the **input language**
  - eliminate left-recursion
  - eliminate left-factors
  - turn every rule into a parsing predicate or (preferably) action.
- describe the static semantics
  - introduce affixes to store information deduced from the input text (symbol table, typing information)
  - insert semantic actions or tests collecting and using this information (for identification, typing, etc.).
- describe the output language
  - deduce a context-free grammar of the **output language**
  - add affixes and semantic actions for generating the output language.

Chapter 7

Multipass transducers

In 6.7 we have shown how to systematically construct a two-level transducer from a given input language to a certain output language. How much work can be done by such a transducer?

Due to the presence of the affix level and the operations, the CDL3 formalism has the computing power of the Turing machine. Anything can be done in one transduction step. From a methodological point of view, the more interesting question is how to program some intended transduction safely and correctly. Rather than doing this in one, maybe very complicated step, in e.g. compiler construction it is good engineering practice to do the transduction in steps, each of which is described by a relatively straightforward transducer. Two techniques for constructing such **multi-pass transducers** can be distinguished:

1. In a **composed transducer**, the output of one transducer (an *intermediate language*) is input to the next.

2. In **coupled transducers**, the output affixes of one transducer are the input affixes of the following transducer. This is the more familiar case (for 'transducer' read 'procedure').

In fact, by considering the output and input languages as special affixes, it can be seen that these two techniques amount to the same thing. The output of one transducer must be reparsed by the next, which may choose to parse it differently than it was constructed.

In this chapter we will deal with multi-pass transducers, introducing a novel mechanism, the **second pass**, which is unique to CDL3.

7.1 Multipass attribute evaluation

The CDL3 executor, among other tasks, keeps track of the calls and the returns of procedures. In fact, the call stack can be seen to contain a representation of (part of) the concrete syntax tree. We will call that representation an **implicit tree** because the programmer can use it without explicitly building it.

Note that the parameters of the production rules are semantic attributes that are evaluated during the parse, from left to right. For evaluating attributes with a complicated interdependence, a single pass over the input text is in many cases not enough, because during the tree decoration one may need information to be extracted from the text that is still to follow. Therefore it is customary to extract an abstract syntax tree in the first pass and then perform various passes over the AST to complete its decoration with attributes.

Although this technique can be expressed prefectly well inCDL3, the language offers another solution for **delayed computations**: during its (first) pass over the input the CDL3 executor keeps the implicit tree for as far as this is needed to perform delayed computations during a second pass.

A general mechanism for delaying computations is quite complicated and costly. However, for many applications, a two-pass approach is adequate:

1. in the first pass, which follows the (concrete) syntax tree, information is collected
2. in the second, which follows the implicit tree, the information collected is used.

In this way, the implicit tree serves instead of the abstract syntax tree, and the programmer is freed from the task of building (and reparsing) the AST.

Many applications fit into this very simple paradigm. That is why we chose to base our language on 2-pass L-attributed Affix Grammars.

We shall introduce the two-pass mechanism in the course of the following example.

7.2 A two-pass transducer

As a running example we will develop a compiler for a simple programming language with block-structure, where names can be applied before being defined. This is a classical problem in compilation because in practically every programming language you find some kind of **forward reference** (e.g. forward goto or block structure).

7.2.1 The input syntax

The concrete syntax of our model language is:

```
program:
   block.
block:
   "begin", units, "end".
units:
   unit, ";", units; unit.
unit:
   application; definition; block.
```

Notice that blocks may be nested.

```
application:
   "apply", identifier.
definition:
   "define", identifier.
```

On this language we shall first impose the classical **context conditions** of block structure, viz.:

1. An identifier may not be defined twice in the same block;
2. Each applied occurrence of an identifier must have some defining occurrence, to be found by searching for the smallest block surrounding the application which contains a definition for that identifier.

Note that according to those rules, an application may come textually before the corresponding definition. The identification can not be performed in one pass over the program, but it can in two.

We shall construct a transducer from this language to some form of code, with a non-trivial semantics.

7.2.2 The input recognizer

The above syntax is LL(1), apart from one left-factor which is easily eliminated, so constructing the input recognizer is straightforward.

```
ACTION program:
   block.
ACTION block:
   "begin", units, "end".
ACTION units:
   unit, units tail.
ACTION units tail:
  ";", units; + .
ACTION unit:
   application; definition; block.
PRED application:
   "apply", identifier.
PRED definition:
   "define", identifier.
```

7.2.3 The environment table

We shall collect the declarations of one block into an affix `RANGE`, which shall be a linear list of definitions (`DEF`s). A `DEF` associates a name (`IDF`) with a blocknumber `BNO` (the nesting level) and an offset `OFFSET` (a number within that block). These attributes will be used in the code generation.

```
RANGE:: empty; RANGE DEF.
DEF:: where IDF has OFFSET in BNO.
IDF:: TEXT.
BNO, OFFSET:: INT.
```

7.2.4 Collecting the declarations

We now extend the input recognizer with semantic actions to compute the `RANGE`s.

```
ACTION program:
  block(0).

ACTION block(>BNO):
  should be token("begin"),
    [ 0 -> OFFSET ],
      [ empty -> RANGE ],
          units(BNO, OFFSET, RANGE),
            should be token("end").

R:: RANGE.
```

```
ACTION units(>BNO, >OFFSET>, >R>):
  application, units tail(BNO, OFFSET, R);
  definition(BNO, OFFSET, R), units tail(BNO, OFFSET, R);
  block(BNO+1), units tail(BNO, OFFSET, R).

ACTION units tail(>BNO, >OFFSET>, >R>):
  is token(";"), units(BNO, OFFSET, R); +.

PRED application:
  is token("apply"),
    ( is identifier(IDF);
      error("identifier expected")).
```

The information available is sufficient to check the first context condition (no multiple declarations).

```
PRED definition(>BNO, >OFFSET>, >R>):
  is token("define"),
    ( is identifier(IDF),
        ( found(IDF, R, OFFSET1, BNO1),
             error("multiple definition for "+IDF);
          [ R where IDF has OFFSET in BNO -> R ],
             [ OFFSET + 1 -> OFFSET ]);
      error("identifier expected")).

TEST found(>IDF, >ENV, OFFSET>, BNO>):
  [ ENV -> conc ENV RANGE ],
    ( found(IDF, RANGE, OFFSET, BNO);
      found(IDF, ENV, OFFSET, BNO) ).
```

In order to also check the second context condition, which may involve forward referencing, we now introduce the mechanism of the second pass.

7.3 The second pass

It is time to introduce the syntax and semantics of the second pass.

7.3.1 Second pass members

Each alternative in a production rule may end with some members that are to be called during the **second pass**. In the text of the alternative, these members are separated from the first pass members by a slash.

7.3.2 Second pass parameters

The second pass members can also use **second pass parameters**, that are parameters whose values cannot be known during the first pass but will be known during the second pass. In the heading of procedures, these parameters are separated from the others by a slash. Of course, the second pass can also make free use of the first pass parameters and local variables as well as the global variables.

Notice that a second pass parameter cannot have the same name as a first pass parameter, so that we cannot overwrite a first pass parameter when coming to the second pass of a procedure.

7.3.3 The environment table

The collection of all declarations which can be applied at a certain point in the program will be found in an environment `ENV`, which is composed out of `RANGE`s.

```
ENV:: conc ENV RANGE; empty.
```

The application rule will, in the second pass, look for a definition in an environment containing all the names available at this place. This environment cannot be known before the end of the analysis because it contains all names defined in the surrounding blocks, therefore in the most external block.

```
PRED application(/ >ENV):
  is token("apply"),
    ( is identifier(IDF)/
        ( found(IDF, ENV, OFFSET, BNO) ;
          error("missing definition for "+IDF+"\n") );
      error("identifier expected")).
```

The previous procedure gets its `ENV` as a second pass input parameter.

```
TEST found(>IDF, >RANGE, OFFSET>, BNO>):
  [ RANGE -> RANGE1 DEF ],
    ( [ DEF ->  where IDF1 has OFFSET in BNO],
        [ IDF1 = IDF ];
      found(IDF, RANGE1, OFFSET, BNO) ).
```

7.4 Order of the execution, affix flow

The second pass of a rule x is executed after the second passes of all the rules x_i that are called by x during its first pass. The second passes of the x_i are executed in the same order as their first passes.

This can be seem restrictive, especially when you want to prepare some values just before calling the second passes of x_i. This can be achieved by calling at the beginning of the first pass a rule with an empty first pass, as in the following procedure.

example

The affix flow between first and second pass (↑) and in the second pass (↓) is shown by the figure:

```
ACTION block      (>BNO     /              >ENV):
  is token("begin"),                         ↓
    [ 0 -> OFFSET ],                         ↓
    [ empty -> RANGE ],                      ↓
    begin of units(BNO/ OFFSET, RANGE, ENV1, ENV),
                          ↑       ↑     ↓
             units(BNO, OFFSET, RANGE/ ENV1),
    should be token("end");
  error("begin expected\n"),
    skip to end.
```

```
ACTION begin of units(>BNO /> OFFSET, >RANGE, ENV1>, >ENV ):
  / [ conc ENV RANGE -> ENV1 ];
```

7.4.1 Interludes

The interludes (see 3.3.4) are executed between the first and the second pass. When there are several interludes in the same or in different modules, the order of their executions is undefined, so that the behavior of an interlude must not depend on the result of another one.

7.5 Failure and second pass

A failure during the first pass indicates that the actual alternative is not the right one. The failure guides the parsing process and therefore the building of the syntax tree. During the second pass only the successful parts of the syntax tree are traversed. In the second pass, no failure is allowed.

Remember that a failure must never give rise to a defect, nor can it cause information to be transferred to another alternative.

7.6 More passes

The second pass elements can be guards (that do not fail) or calls of functions or actions. These called rules can call some other rules, building the tree of the calls needed to compute the second pass results. The resulting implicit syntax tree has the same status as the normal one and can be travelled a second time. This is simply done by calling during the second pass rules that have a second pass. This is not really a third pass on the syntax tree but a second pass on a subtree that was not built during the first pass, and therefore that was not ready for the normal second pass.

7.7 Generating code

We now extend our example transducer to generate instructions for an abstract machine, dealing with a simple form of storage management. It has the following instructions:

`reserve bn, length`	allocate a storage area of length `length` for block number `bn`
`apply bn, offset`	access the storage location corresponding to the variable in block `bn` with offset `offset`
`unreserve bn, length`	release the storage area with this `bn` and `length`.

Although not very realistic, these instructions are meant to capture the essence of storage management.

We first define the output language to which we want to transduce (on the affix level):

```
CODE :: empty; CODE INSTR.
INSTR :: reserve BNO OFFSET;
         apply BNO OFFSET;
         unreserve BNO OFFSET.
```

We now add the second-pass affix `CODE` to those procedures that will generate one or more `INSTR` codes.

7.8 Complete example

Using the module lexico shown in chapter 3.3, the root module is the following

```
ROOT program.
USES lexico.
RANGE:: empty; RANGE DEF.
R=RANGE.
DEF:: where IDF has OFFSET in BNO.
BNO, OFFSET:: INT.
IDF:: TEXT.
ENV:: conc ENV RANGE; empty.

CODE :: empty; CODE INSTR.
INSTR :: reserve BNO OFFSET;
         apply BNO OFFSET;
         unreserve BNO OFFSET.

ACTION program:
  [ empty -> CODE ],
    [ empty -> ENV ],
    block(0/ ENV , CODE).

ACTION block(>BNO/ >ENV , >CODE>):
  is token("begin"),
    [ 0 -> OFFSET ],
      [ empty -> RANGE ],
      begin of units(BNO/ OFFSET, RANGE, ENV1, CODE, ENV),
               units(BNO, OFFSET, RANGE/ ENV1, CODE),
      should be token("end")/
          [ CODE unreserve BNO OFFSET -> CODE ];
  error("begin expected\n"),
    skip to end.

ACTION begin of units(>BNO
                      /> OFFSET, >RANGE, ENV1>, >CODE>, >ENV):
  / [ conc ENV RANGE -> ENV1 ],
    [ CODE reserve BNO OFFSET -> CODE ].

ACTION units(>BNO, >OFFSET>, >RANGE>/ >ENV, >CODE>):
  application(/ ENV , CODE),
    units tail(BNO, OFFSET, RANGE/ ENV, CODE);
  definition(BNO+1, OFFSET, RANGE),
    units tail(BNO, OFFSET, RANGE/ ENV, CODE);
  block(BNO+1/ ENV , CODE),
    units tail(BNO, OFFSET, RANGE/ ENV, CODE).

ACTION units tail(>BNO, >OFFSET>, >R>/ >ENV , >CODE>):
  is token(";"),
    units(BNO, OFFSET, R/ ENV , CODE);
  + .
```

```
PRED application(/ >ENV , >CODE>):
  is token("apply"),
    ( is identifier(IDF)/
        ( found(IDF, ENV, OFFSET, BNO) ,
           [CODE apply BNO OFFSET ->  CODE];
          error("missing definition for "+IDF+"\n") );
      error("identifier expected")).

PRED definition(>BNO, >OFFSET>, >R>):
  is token("define"),
    ( is identifier(IDF) ,
         ( found(IDF, R, OFFSET1, BNO1),
             error("multiple definition for "+IDF+"\n");
          [R where IDF has OFFSET in BNO -> R],
             [ OFFSET+1->OFFSET] );
      error("identifier expected")).
```

7.9 conclusion

For many transductions the two-pass paradigm (first pass: recognize; second pass: generate) is quite adequate. For others it may be insufficient. There may still be situations where an explicit syntax tree is necessary, for instance when we want to compare two parts of the parse tree. A typical example is type checking where we have to compare the declaration of objects with the instructions where they are used. In this case, we must build during the first pass an abstract tree of the declarations known as the **symbol table**.

Chapter 8

Static semantic checks

In this chapter we discuss one of the more interesting aspects of the CDL3 technology, viz. the semantic checks performed by the compiler. This chapter is a rewritten and expanded version of [FEU78] on CDL2.

8.1 Introduction

By a **static semantic check** we mean the investigation of the **plausibility** of a syntactically correct program, based on an analysis of the flow-of-control and flow-of-data in the program, verifying those semantic properties of the program that can be verified statically, i.e. by inspecting the program without actually executing it.

This check may (classically) answer such questions as:

- do all applied identifiers have a defining occurrence?
- are, in all assignations, the types of source and destination compatible?
- is the value referred to by a reference compatible with the use made of it?

but for not so classical languages it may also determine:

- do all variables have defined values upon being used?
- are various parts of the program free from side effects upon one another?
- is the (concurrent) program guaranteed to be free of deadlocks?

and other questions pertaining to the execution of the program but (under specific conditions) answerable at compile time.

Such a static semantic check attempts to catch semantic errors which would otherwise be detected only dynamically, with the intention of

- shortening the test phase and the whole implementation process
- increasing the reliability of the product, and
- improving maintainability.

In the extreme case, static semantic checking should eliminate all incorrect programs, but we will argue that even a partial check can already be of enormous value.

In this chapter, we will be concerned with the feasibility of static semantic checking, and with those aspects of the design of a language that affect the effectiveness and the efficiency of such a check.

In particular, we will discuss the question how such a check can be made for an language, i.e. a language which allows in its programs the application of algorithms, objects and types defined outside the framework of the language itself. The techniques described here are not new, but both their actual feasibility and the particular design choices made in the implementation language CDL3 may be of interest.

8.2 Semantic checks

We are concerned with the investigation by the compiler of the plausibility of syntactically correct programs, taking into account only those properties of the program which can be analyzed statically, i.e. without actually executing it.

Such a check is by no means the same as a proof of the correctness of the program, which would necessitate checking the program against some specification. In most cases, no suitable specification is available. The program can however be checked against itself: it can be checked for the absence of contradictions and blatant impossibilities, by exploiting the redundancy present in programs. What then constitutes useful redundancy?

8.2.1 Useful redundancy

The prime example of **redundancy** in programming languages is the presence of mandatory declarations, including type specifications.

Although such declarations are technically not necessary (declarations by default as in FORTRAN or dynamic typing as APL can be implemented just as well) they provide for redundancy, allowing:

- the detection of most spelling errors in identifiers which, in languages with default declarations, might lead to surprising program behaviour whose cause may be hard to find by testing
- the static attribution of a type to objects, allowing the prohibition of assignations to unsuitable variables and of the applications of unsuitable operations.

That is, declarations introduce useful redundancy. Redundancy is not useful when it cannot be checked, (like e.g. comments), when it is not mandatory (like the optional **goto** symbol in ALGOL 68) or when it serves merely to check whether the programmer at one place in his program can repeat what he has said at another, as e.g. the destination list in an assigned goto in FORTRAN.

Redundancy is useful when it serves to specify abstractly the *intention* of the programmer, in such a way that it can be checked against his concrete actions elsewhere in the program.

In designing a programming language, useful redundancy should be introduced systematically. As an example, we will discuss the redundancies introduced into CDL3.

In the next sections we will discuss some limitations and problems in static checking.

8.2.2 Incompleteness of a static semantic check

A static semantic check can in general not be complete: the compiler knows only the static properties of the program and will therefore have to make worst-case assumptions ("if something can go wrong, it will").

In the first place, the knowledge of the flow-of-control may be insufficient. As an example, consider the following piece of program:

if $(p * p)$ **mod** $4 = 2$ **then**
  begin var a : **real**; {*uninitialized*!}
  print(a) **end**

It is not reasonable to demand from the compiler that it knows that the square of an integer number modulo four can assume only the values 0 or 1 (did *you* know it?), and that therefore the condition always yields false. The semantic check should report a potential error, even though the clever programmer might have known that the error cannot occur.

In competently written programs, such cleverness should be infrequent, and since the behaviour of programs depends critically on their input, we will have to assume that all paths through a program are taken, and therefore have to report all potential errors. We will take the point of view that programs which are not demonstrably correct are in error. This may be unjust, especially for some tricky programs. But from an engineering standpoint it makes good sense, since a program which is not obviously correct may be a source of pitfalls in maintenance. A serious program, like Caesar's wife, should be above suspicion.

Apart from the lack of knowledge about the dynamic flow of control, another reason for incompleteness of static semantic checks may be the lack of knowledge about the identity of variables.

In order to answer e.g. the question "does each variable have a defined value upon being used", the compiler has to know what variables are assigned to. We will term a variable x an **alias** for a variable y if an assignment to y affects the value of x, as in the ALGOL 68 program

begin real x;
  procedure $f(y)$; $y := 0$;
  procedure p; $f(x)$;
  procedure $t(z)$; z;
  $t(p)$;
  print(x)
end

Whether questions about identity of variables are statically decidable depends on the presence in the language of such alias-makers as the FORTRAN `EQUIVALENCE` statement, the PASCAL pointer assignment or the ALGOL 68 identity declarations of the form

ref amode $x = y$

To make matters worse, aliasing may be dynamic as in

ref amode $x = y1[i]$

making it under circumstances impossible to decide statically whether the array $y1$ has been properly initialized. The systematic avoidance of any dangerous aliasing has been the primary design goal of the language EUCLID [LAM77]. In [LAN73] it is discussed in the context of ALGOL 60 how the presence or absence of certain language properties may affect the static decidability of specific properties of programs; in particular this article shows the necessity of full specification of formal parameters.

8.2.3 Incompleteness caused by open-endedness

Open-endedness of the language is another cause of incompleteness of the static semantic checks. Fundamentally, the semantics of borrowed algorithms is outside the scope of the check, and any knowledge about the effect of such algorithms has to be introduced explicitly by a specification which is the responsibility of the programmer. Without such a specification, it would be impossible to write meaningful programs. The correctness of the specification itself cannot be decided within the framework of the language, but has to be taken on trust.

As an example, the specification of a library routine

proc (**real**) **real** *sin* = **fromlib**("ASIN")

allows a modicum of security in calling that routine, provided the specification adequately and correctly captures the relevant properties of the corresponding library routine. It is definitely more revealing than just

```
EXTERNAL FUNCTION ASIN
```

or even

```
EXTERNAL ASIN
```

In designing an open-ended language, the necessary redundancy has to be carefully tailored in, in order to minimize the unavoidable risks.

Specifications of a borrowed algorithm may include:

- its logical effect, given in the form of logical relationships or algebraic axioms (not pursued in this paper)
- the type of its result
- the number, order and types of its parameters
- its calling mode (e.g. an algorithm borrowed form PL/1 may have to be called differently than one borrowed form LISP)
- the presence or absence of side-effects.

The obvious way to include such specifications into a language is to demand the writing of a heading for each algorithm to be borrowed.

Whether the semantics of the algorithm borrowed fits the heading is lastly the responsibility of the programmer writing the algorithm, who will have to program in a disciplined fashion.

Open-endedness should be used to introduce a small number of primitives, whose relevant properties are specified as precisely as possible, and *not* for the promiscuous spread of code inserts all over the program, demanded by ill-informed assembly programmers as a precondition for reluctantly writing their programs in a systems implementation language.

8.3 Static semantic checking in CDL3: a case study

The basis for the semantic checking possibilities in CDL3 are the concise specifications of attributes of each algorithm with regards to its result, its global effect and its parameters.

8.3.1 Specification of the result

The body of an algorithm may be structured such that the algorithm always returns *true* (it always *succeeds*), or it may be such that it can also return *false* (it may *fail*). The programmer specifies for each algorithm whether it (according to her) may fail. On the basis of this redundancy, a double check is made:

- The body of each procedure is checked against its specification
- All calls of a procedure are checked against its specification.

8.3.2 Specification of the global effect

Let us define as the **environment** of a program the set of all its global variables, arrays and input/output files. A change to this environment is termed an *activity*. Any algorithm may perform an activity upon success, a so-called **effect**, or upon failure, which we then call a **defect**. As a consequence of the idea of considering all procedures as parsing procedures, we consider defects as unwanted side-effects. In CDL3 defects are forbidden and effects must be specified explicitly.

In connection with the result attribute, we distinguish four classes of algorithms:

test: failure possible, no effect

predicate: failure possible, effect

function: failure impossible, no effect

action: failure impossible, effect

8.3.3 Specification of parameters

In CDL3, all *parameters* are passed via a copy/restore-mechanism, i.e. on each call, local variables are set up, which may be initialized with the value of the corresponding actual parameter (*copy*) and whose value may be passed to the corresponding actual parameter after elaboration of the procedure (*restore*). The restoring takes place only after successful elaboration of the algorithm. Upon failure, no restoring is done.

Each parameter is specified to have one of the following four directions:

inherited: copied, not restored

derived: not copied, restored upon success

transient: copied, restored upon success

free: not copied, not restored (local variable).

It should be noted that in CDL3 only simple variables and constants can be used as actual parameters: neither arrays nor procedures can be passed as parameters. These limitations ensure that the alias problem for simple variables is decidable.

8.3.4 Visibility rules

There are just two scope levels in CDL3 on which affixes may be declared. On the *module level* the (global) variables, constants and arrays are declared, whose scope is the module.

On the *algorithm level*, affixes are declared whose scope is the algorithm. There is no nesting of procedure declarations, no passing of procedures as parameters, and no block structure within procedures.

8.4 Local semantic check

The **local semantic check** deals with one procedure at a time, checking its body against its heading, i.e. checking the programmer's actions against his intentions.

The body of the procedure is represented by its *control graph* (1.2.5). For every affix and every node of the graph an *abstract value* is calculated denoting properties of the values the affix may take at execution time upon reaching the node. The environment and changes to that environment are modeled by one pseudo affix with the value *global*.

8.4.1 The control graph

The nodes of the **control graph** are the members of the body of the procedure, i.e. calls for algorithms, guards and control operators. If the member may succeed of fail, two edges leave the member, on labelled t and one labelled f (the *true-edge* and the *false-edge*). If the member cannot fail, it is only left by a true-edge. All members of one alternative are interconnected sequentially by their true-edges, while each false-edge points to the first member of the next alternative. The true-edge of the last member of each alternative points to the *true exit*, while the false-edges of all members of the last alternative point to the *false exit*. This structure may be modified by the use of the control operators for repetition and leave and by grouping.

Each member corresponds to a node in the graph. The node corresponding to the member reached upon entering the rule is called the *entrance*. The *true-successor* of a member m is the node pointed to by m's true-edge, its *false-successor* is the node pointed to by its false-edge.

8.4.2 Abstract values

The **abstract value** of an affix denotes properties of the set of values the affix may take at execution time. The properties we are interested in are:

- is the value *undefined*?
- is the value obtained by copying the actual parameter before elaboration of the rule (value *inherited* from caller)?
- is the value obtained by some call in the body of the algorithm (value *derived* from called), and if so, by which call?

Since some paths through the program graph may join, more than one of the cases above may in general occur at a node. Thus the abstract values are modeled by sets which are constructed by union from the following elementary abstract values:

$\{undef,\ inherited,\ der_m \mid m \text{ member of the procedure}\}$

We say a value is *assigned* to an affix a by some member m if m has that affix at a derived or transient position. A value is assigned to the pseudo affix modeling the environment if m has some global variable at a derived or transient position or if m is a call for an action or predicate (thus, all global variables are lumped together).

An affix is *used* by a member m if m has the affix at an inherited or transient position. An affix is used by the true exit if it has the type *derived* or *transient*. The pseudo affix is used by every node. The calculation of the abstract values AV may then proceed iteratively as follows, using well-known techniques (see e.g. [COU77]).

1. For all nodes n and for all affixes a: $AV(a,n) := \emptyset$.
2. For all affixes a with type free or derived: $AV(a, entrance) := \{undef\}$

   For all affixes a with type inherited, transient and global:

   $AV(a, entrance) := \{inherited\}$.
3. We now have to walk through the graph starting at the entrance and performing calculations of the abstract values at each node. In the case of loops, parts of the graph will have to be considered more than once; the walk ends, if in a step no abstract value is changed. Termination of the algorithm is guaranteed by the fact that abstract values are only changed by the addition of elements.

   Upon passing a member m the following calculations have to be done for every affix a:

   $AV(a, falsesucc(m)) := AV(a,m) \cup AV(a, falsesucc(m))$

   a is not assigned to by m

   $AV(a, truesucc(m)) := AV(a,m) \cup AV(a, truesucc(m))$

   a is assigned to by m

   $AV(a, truesucc(m)) := der_m \cup AV(a, truesucc(m))$

8.4.3 Conditions on the use of abstract values

Using the abstract values calculated above and some additional information obtainable from the graph, the following conditions are checked:

- no undefined value may be used, neither by using it as input parameter to a call nor by restoring it upon success. If a node n uses an affix a, then

  $\{undef\} \cap AV(a,n) \stackrel{!}{=} \emptyset$

  The sign $\stackrel{!}{=}$ is to be read as "must be", and $\stackrel{!}{\neq}$ as "must not be".
- every value that is not undefined should be used somewhere, otherwise its computation was superfluous.

  We may model this by constructing for each affix a *possibly used abstract value PAV*, which is the union of all abstract values of the affix omitting $\{undef\}$. Then we can construct the *used abstract value UAV* as the union of all abstract values which are used.

  $PAV(a) := \cup_n(AV(a,n)) \setminus \{undef\}$

  $UAV(a) := \cup_n(AV(a,n)) \mid n$ uses a

  $PAV(a) \stackrel{!}{=} UAV(a)$

- no value, that has been created by an assignment in one alternative, may be passed to any alternative following it (*independence of alternatives* within a procedure body).

  $AV(a, n) \cap \{der_m \mid m \text{ is in a previous alternative}\} \stackrel{!}{=} \emptyset$

- a transient affix must obtain a value somewhere. For all affixes a with type *transient*:

  $AV(a, \mathit{trueexit}) \stackrel{!}{\neq} \{\mathit{inherited}\}$

- no activity may be performed upon failure (defect). For the pseudo affix *global*:

  $AV(\mathit{global}, \mathit{falseexit}) \stackrel{!}{=} \{\mathit{inherited}\}$

- if the procedure has one of the types `ACTION` or `PREDICATE`, it must have an effect. For the pseudo affix *global*:

  $AV(\mathit{global}, \mathit{trueexit}) \stackrel{!}{\neq} \{\mathit{inherited}\}$

- if the procedure has one of the types `FUNCTION` or `TEST`, it must not have an effect.

  $AV(\mathit{global}, \mathit{trueexit}) \stackrel{!}{=} \{\mathit{inherited}\}$

8.5 Global semantic check

Due to the large amount of time and space which would be necessary for a complete analysis, the control over the use of global variables has not been made as tight as the control over affixes. It is NOT YET checked, whether undefined values of global variables may be used somewhere.

8.5.1 Control over initialization

Some languages (e.g. BCPL) force the programmer to initialize all global variables when declaring them. For CDL3, this does not seem to be the right way, since the variables may then only be initialized by constants or by previously defined variables.

Firstly, this is sensitive to the specific order of the declarations, which we would like to avoid. Secondly, even though for some kinds of variables the static initialization may work (initialize with zero or nil), for others they make no sense (how to initialize a global variable that is to hold some attribute of the current element of a huge data structure?).

Even if in such a case the programmer initializes the variable with an "impossible" value and checks for that, he has to perform run-time tests in order to find out if there exists some path in his program leading to the use of this "impossible" value. The compiler can carry a big part of this work on its shoulders, by testing statically whether a variable may be used without initialization — provided the language does not enforce static initialization in those cases where this makes no sense.

8.5.2 Applicability to CDL3

CDL3 has some properties that simplify a global check:

- the impossibility of exporting variables from one module to another decreases the number of global variables to be treated at any one time.

- most of the variables — at least those used throughout the program — are initialized in the prelude part and thus need not be considered in the main part of the program.

- the control structures of CDL3 are such that the control graph is **reducible** [ALL70, LAM77] and therefore easily analyzable and optimizable.

- the structure of CDL3 programs as a hierarchy of small procedures gives rise to a rather efficient "Bottom-Up" analysis hardly applicable to languages with a more complicated structure.

- due to the copy/restore parameter mechanism and the absence of formal procedures, the alias problem is for simple variables decidable, so that the flow of data can easily be analyzed.

8.5.3 The method

For every procedure, it is possible to calculate the set of variables that must be initialized before calling it. Let us call this set the *use* of the procedure. It is also possible to calculate the set of variables that are initialized by the procedure: this is the set of global variables that obtain values on all possible paths through its body to the success exit, but which are not contained in its use. This set is called the *yield* of the procedure.

If we calculate these two sets for all procedures, including a quasi-procedure constructed from the prelude, root and postlude-calls of the program, all variables which are in the use of the prelude are used somewhere without initialization.

We can improve this algorithm by performing the calculation just for the first prelude call and then disregarding the variables which are in the yield of that prelude call. Repeating these steps for all prelude, root and postlude-calls, we get a decreasing number of variables for each step. Since our experience shows that most of the global variables are initialized in the prelude-part, the analysis of roots, which contain most of the procedures, does not have to consider many variables.

The global check could be extended to keep track of not only the defined and undefined values, but also of the range of a (defined) value. The mechanism should not be greatly different (see [COU77, KIL73]).

8.5.4 Separate compilation

Within the framework of our checks, it is especially interesting if some exported algorithm uses variables, which have to be initialized by either another exported algorithm or by some prelude or root call. The conditions that have to be fulfilled between modules are simple as long as no mutual intermodule communication is allowed, e.g. if the modules form a hierarchy. Having completely unrestricted communication between modules, the conditions obtain such a complexity that they may hardly be checked.

As long as we compile the importing and exporting modules together, it is possible to check globally that no uninitialized variables are used.

When separately compiling these two modules we have a problem — in compiling the defining module we do not know the order of the calls; in compiling the applying module we do not know the restrictions on the order of calls. This problem may be aleviated by transforming the global variables to local parameters. The normal static semantic checks can then be applied to ensure the globals are properly initialized. The major drawback here is that changing the globals of a module might trigger the analysis of other modules that are seemingly unrelated.

8.6 Semantic check results

In using implementation languages, the presence of a static semantic check can catch a great many errors which would otherwise be found only by run-time tests.

Of the semantic checks for CDL3 described here, the local check has been in use since early 1976. The global check is in use in the CDL2 LAB, where the whole program is accessible.

After initial resistance from our (quite conservative) users, and improvements to its human engineering, the semantic checker has come to be accepted as an extremely useful production tool. One reason why each CDL3 program is now routinely checked, is the speed at which the check is performed. The local check needs about 16 seconds of CPU time for a 5000 lines program on an IBM/370-158. This contrasts favourably to the 0.3 to 0.5 seconds per FORTRAN statement mentioned for a similar (but weaker) checker implemented on a CDC 6400 by Fosdic and Osterweil [FOS76], which in that form seems hardly applicable to large programs.

The methods described are of value to implementation languages in general. Some properties seem to be essential for the application of the checks:

- the copy/restore parameter mechanism and the absence of procedures as parameters are necessary (even though not sufficient) for solving the alias-problem and simplifying the checks on data flow
- tight visibility control and a simplified block structure enhance the perspicuity of the program and reduce the complexity of the calculations involved in the checks
- in designing an open-ended implementation language, useful redundancy should carefully be introduced.

Appendix A

Syntax of CDL3

In this chapter a Context-Free grammar of CDL3 is given, introducing terminology for speaking about the most important constituents of a program. In the form of remarks between the syntax rules, the semantics of the various constructs is discussed.

A.1 Syntactic abbreviations

In order to keep this syntax short and concise, we make use of a number of conventions for omitting redundant rules. For example, in the grammar we shall need the concept of a list of module names, with a syntax rule like

```
module-name-list:
  module-name;
  module-name, comma-token, module-name-list.
```

But we shall also have member-list, with a similar rule; and there will be still more forms of lists. We therefore introduce the convention that for any notion of the form N-list (in which N stands for some word) we can assume

a1) N-list: N; N, comma-token, N-list.

Applied to the word member, this leads to

```
member-list:
  member;
  member, comma-token, member-list.
```

Of course this is just a device to increase the abstraction level and enhance the readability of the grammar.

Further abbreviations are:

a2) N-option: N; .

Something that is optional may be left out.

a3) N-sequence: N; N, N-sequence.

A sequence of things consists of one or more of those things, one after another. Notice the difference: in an N-list there is a separator, the comma-token, between two consecutive elements, but there is no separator in an N-sequence.

a4) N-pack: open-token, N-list, close-token.

A pack is a list enclosed between parentheses.

In this appendix the names of syntactic constructs will be written in sans serif font, using some spelling variations (like primaries rather than primarys) for linguistic reasons.

A.2 Programs and modules

A program consists of zero or more modules, followed by the main-module. Each module may use objects defined by previous modules and may define objects for use in subsequent modules.

program: module-sequence-option, main-module.

A module consists of a module-body, preceded by a module-head and module interface.

module:
module-head, module-interface, control, module-body.

The module-head specifies the name of the module and its module variables. Due to the current language implementation the module name must be restricted to at most 8 characters.

module-head:
"MODULE", module-name, module-variables-option, ".".

The module-variables, if any, are specified as one special affix alternative. The variables mentioned in this part are visible in the whole module.

module-variables: "=", affix-alternative.

The main-module of a program is its last module, the one that contains the root of the program.

main-module: root, uses-part-option, module-body.

The root must be a nonterminal with arity zero that must be defined later in the main-module.

root: "ROOT", nonterminal-name, ".".

A.2.1 Module interface

The module-interface consists of two parts, each of which can be empty.

module-interface: defines-part-option, uses-part-option.

The defines-part comprizes a specification for every object, defined in the module, that is exported to subsequent modules.

defines-part: "DEFINES", object-specification-list, ".".

object-specification: affix-name; procedure-head.

The uses-part gives the names of all modules whose exported objects are imported in this module.

uses-part: "USES", module-name-list, ".".

A.2.2 Module control

The control gives a list of procedures that are to be executed at a specific moment.

control:
  control-procedure-sequence-option.

control-procedure:
  control-procedure-head, ":", alternatives, ".".

control-procedure-head:
  control-type, nonterminal-name.

control-type:
  "PRELUDE";
  "INTERLUDE";
  "POSTLUDE".

A prelude (resp. interlude,postlude) is a special action that is executed at the start of the execution (resp. between the first and second pass, at the end of the execution).

A.2.3 Module body

The body of a module consists of a sequence of definitions. Their order is irrelevant.

module-body: definition-sequence.

definition: procedure; affix-rule.

A.3 Procedures

The definition of one nonterminal name (*nonterminal* for short) consists of one or more procedures with that nonterminal as head. The various procedures making up one definition need not have the same *arity* (= number of parameters).

procedure: procedure-head, ":", alternatives, ".".

A.3.1 Rule head

procedure-head:
  procedure-type, nonterminal-name, formal-parameter-pack-option.

The procedure-type defines whether a procedure can fail (test or predicate) or not (function or action) and whether it has global effects (predicate or action) or not (test or function).

procedure-type:
  "ACTION";
  "PRED";
  "FUNCTION";
  "TEST".

The character(s) > indicates the direction of each formal parameter.

formal-parameter:
  input-parameter;
  output-parameter;
  transient-parameter.

input-parameter: ">", variable.

output-parameter: variable, ">".

transient-parameter: ">", variable, ">".

A.3.2 Alternatives

The alternatives of a procedure are tried in textual order until the success of one of them or the failure of all of them.

alternatives:
  alternative;
  alternative, ";", alternatives.

alternative:
  member-list, second-pass-option;
  second-pass;
  succes-operator;
  member-list, ",", group;
  member-list, ",", operator.

The members of an alternative and the optional group or operator are tried in textual order until the success of all of them or the failure of one of them.

group:
  "(", alternatives, ")".

A group is in fact a shorthand for the introduction of an anonymous new procedure. An alternative may specify a second-pass.

A.3.3 Second pass

The execution of a second-pass is delayed until the end of the first pass of the root. It must not fail.

second-pass:
  "/", member-list, second-pass-group-option.

second-pass-group:
  "(", second-pass-alternatives, ")".

second-pass-alternatives:
  second-pass-alternative;
  second-pass-alternative, ";", second-pass-alternatives.

second-pass-alternative:
  member-list, second-pass-group-option;
  member-list, operator;
  succes-operator.

A.3.4 Members

member:
  call;
  guard.

call: nonterminal-name, parameter-pack-option.

parameter: variable; affix-expression.

When a parameter is an affix-expression it implies a join or a split.

operator:
  failure-operator;
  success-operator;
  abort-operator;
  exit-operator.

failure-operator: "–".
success-operator: "+".
abort-operator: "?".
exit-operator: "!".

A.3.5 Guards

guard: "[", confrontation, "]".

confrontation:
  join;
  split;
  assign;
  equal.

join:
  affix-expression, "->", variable.

A join gives a value to a variable, this value is a tree built according to the affix expression.

split:
  variable, "->", affix-expression.

A split succeeds if and only if the value of its variable conforms to the tree pattern given by the affix expression; upon success it gives values to the variables appearing in this affix expression.

assign:
  variable, "->", variable.

equal:
  variable, "=", variable.

An assign copies a value into another variable while an equal tests the equality of two variables.

A.3.6 Affix expressions

An expression denotes a tree-pattern, which can be used to build (join) or to split a tree (split).

affix-expression: term-sequence.

term:
  variable;
  meta-terminal;
  arithmetic-expression;
  text-expression.

arithmetic-expression:
  arithmetic-term;
  arithmetic-term, arithmetic-operator, arithmetic-expression.

arithmetic-term: variable; number.

arithmetic-operator:

"+";	*addition*
"−";	*substraction*
"*";	*multiplication*
"%";	*division*
"%%";	*modulo,*
"<<";	*left-shift,*
">>";	*right-shift*
"&";	*bitwise and*
"|";	*bitwise or*
"^".	*bitwise exclusive or*

The arithmetic-operators are all dyadic: $INT \times INT \rightarrow INT$.

text-expression:
  text-term;
  text-term, "+", text-expression.

Here the + denotes the concatenation of texts.

text-term: variable; text-constant.

A.4 Metarules

A metarule serves to specify some *domain* as the disjunct union of the domains of its meta-alternatives and binds it to one or more affix names specified in its meta rule head.

meta-rule: meta-rule-head, "::", affix-alternatives, ".".

meta-rule-head: affix-name-list.

affix-alternatives:
  affix-alternative;
  affix-alternative, ";", affix-alternatives.

affix-alternative: meta-element-sequence.

Each affix alternative specifies a tree-pattern, a tuple of meta-elements.

meta-element: affix-name; meta-terminal.

A.5 Lexical conventions

A.5.1 Names

module-name: letter-sequence.

nonterminal-name: name.

affix-name: capital-letter-sequence.

The optional number in a variable serves to distinguish different instances of affixes with the same domain.

variable: affix-name, number-option.

Names of nonterminal are formed from small letters, and may contain embedded spaces.

name:
  name, embedded-space, small-letter-sequence;
  small-letter-sequence.

The two forms of embedded space are equivalent.

embedded-space: "_" ; " ".

letter: small-letter; capital-letter.

small-letter: "a"; "b"; ...; "z".

capital-letter: "A"; "B"; ...; "Z".

A.5.2 Constants

number: digit-sequence.

digit: "0"; "1"; "2"; "3"; "4"; "5"; "6"; "7"; "8"; "9".

text-constant: double-quote, string-without-quotes, double-quote.

Text constants are written between quotes. Within a quoted text, a quote can be denoted by `\"`. The sequence `\n` denotes an end-of-line character, a backslash is denoted by `\\`.

meta-terminal: small-letter-sequence.

A.5.3 Comments

A comment starts with a comment-symbol # and comprizes the rest of the input line or upto the next comment-symbol on the same input line. As usual, comments carry no meaning, as well as extra spaces and carriage returns..

Appendix B

The predefined module predef.k3

The module whose defines-part follows is included whenever a program is compiled (no USES specification is needed). The procedures described here are in fact written in C using the standard C-libraries. Refer to their documentation for more details.

```
FORMAT :: FORMAT TEXT;
          FORMAT INT;
          TEXT;
          INT.
```

You can work on any number of input or output files. The text given when opening a file is its pathname following the Unix notation.

```
TEST open input file (>TEXT,FILE>)
PRED read line (>FILE,TEXT>)
PRED read char (>FILE,TEXT>)
PRED read char (>FILE,INT>)
```

The read line routine reads a line from the FILE. Reading stops after an end-of-file marker or a newline. If a newline is read, it is stored in the TEXT. The read char reads a single character from the FILE. Depending on the type of the second parameter is returned as a TEXT or INT.

The formatted version of write takes a list of integers and texts and writes them on the given file.

```
TEST open output file (>TEXT,FILE>)
ACTION write (>FILE,>TEXT)
ACTION write (>FILE,>INT)
ACTION write char (>FILE,>INT)
ACTION formatted write (>FILE,>FORMAT)
FUNCTION formatted trace (>FILE,>FORMAT)
```

The trace function writes on a file but its hides effect (function instead of action) so that you can use it for tracing alternatives that will fail afterwards.

```
ACTION close (>FILE)
PRED seek (>FILE,>INT)
```

The standard files are implicitly used by these functions:

```
PRED open standard output (FILE>)
```

```
PRED open standard error (FILE>)
PRED open standard input (FILE>)

PRED read line (TEXT>)
PRED read char (TEXT>)
PRED read char (INT>)
ACTION write (>TEXT)
ACTION write (>INT)
FUNCTION trace (>TEXT)
FUNCTION trace (>INT)
```

The trace functions hide their effect on the standard error.

Arithmetic comparisons:

```
TEST less (>INT,>INT1)
TEST lesseq (>INT,>INT1)
TEST greater (>INT,>INT1)
TEST greatereq (>INT,>INT1)
```

Text operations

```
FUNCTION length (>TEXT,INT>)
FUNCTION extract (>TEXT,>INT1,>INT2,TEXT1>)
TEST is subtext (>TEXT,>INT>,>TEXT1)
TEST asciicode (>TEXT,>INT1,INT2>)
```

The extract function puts into TEXT1 the INT2 - INT1 characters starting at position INT1 in TEXT. If INT2 $\leq$ INT1, an empty text is returned. The test is subtext looks whether TEXT at position INT contains TEXT1. Upon success it adds the length of TEXT1 to INT. The test asciicode returns the ascii value of the character at position INT1 in TEXT in INT2 if this character is present.

The following procedures test a character in a text.

```
TEST between (>TEXT,>INT,>TEXT1,>TEXT2)
TEST not between (>TEXT,>INT,>TEXT1,>TEXT2)
```

Both between and not between check if character INT in TEXT is between respectively not between the characters in TEXT1 and TEXT2. The ordering is according the ASCII table. The first character in a TEXT has index 0.

Texts may be compared with

```
TEST before (>TEXT,>TEXT1)
TEST is prefix (>TEXT,>TEXT1,TEXT2>)
TEST prefix (>TEXT,>INT,TEXT1>,TEXT2>)
```

is prefix looks for TEXT at the beginning of TEXT1. If it is present the remainder is put into TEXT2. prefix cuts TEXT into two parts: INT characters are put into TEXT1, the remainder is put into TEXT2.

The following procedures convert INTs in TEXTs and vice versa.

```
FUNCTION bindec (>INT,TEXT>)
TEST decbin (>TEXT,INT>)
```

The following procedure is useful to create hash indexes in arrays.

```
FUNCTION hash (>TEXT,>INT1,INT2>)
```

The function hash gives to INT2 a value between 0 and INT1–1, depending on the TEXT.

The following procedures are standard Unix functions.

```
TEST command arg (>INT,TEXT>)
FUNCTION exit (>INT)
TEST get from environment(>TEXT1,TEXT2>)
FUNCTION time (INT>)
PRED execute (>TEXT)
TEST file date (>TEXT,INT>)
```

The test command arg gives, if it exists, the INTth argument of the command line. The function exit exits with return code INT. The test get from environment gets the value (if defined) of the environment variable whose name is TEXT1 in the Unix environment. The function time returns the current time in seconds since 1 January 1970. The predicate execute hands its argument to the command interpreter sh for execution. The test file date gives the date of last modification of a file (if it exists).

Appendix C

Exercises

C.1 Exercise 1

purposes:

- get acquainted with the local implementation of CDL3
- get experience in programming-in-the-small

Exercise 1a

input: a stream of characters, ending on a period
task: copy input to output, up to and including the period.

From the predefined procedures you can use:

- `PRED read char(TEXT>)` that gives one character read from the standard input
- `ACTION write char(>TEXT)` that writes one character on the standard output

Exercise 1b

input: a stream of characters, ending on a period
task: copy input to output in reverse order, up to and including the period.

Do not make use of global variables or composed types.

C.2 Exercise 2

purposes:

- further experience in programming-in-the-small
- first experience in programming-in-the-large.

Exercise 2a

Write a procedure with two integer parameters `print (>INT, >WIDTH)` that prints `INT` in `WIDTH` columns, with a minus-sign in the first column iff `INT` is negative. Overflow of the width will have to be dealt with (it is better to spoil the layout than to loose information).

Examples of the intended output (width=3), with vertical bars inserted to delimit the three columns:

```
   +3   |  3|
   -3   | -3|
 +114   |114|
+9276   |927|6
 -987   |-98|7
```

In spite of the fact that this is a very small exercise, please implement the program as two modules, `printing` and `driver`.

```
-------------
|           |
|  driver   |
|           |
-------------
      |
      |
-------------
|           |
| printing  |
|           |
-------------
```

Exercise 2b

A number consists of one or more digits, possibly preceded by layout characters.

Add to the program a module `reading`, implementing buffered reading of lines from the keyboard, exporting at least the following operations:

```
TEST ahead (>TEXT)
PRED is (>TEXT)
ACT  should be (>TEXT)
PRED is number (VAL>)
PRED is plus
PRED is minus
ACT  skip layout
```

Exercise 2c

Extend the test driver so that it forms a desk-calculator of your own design, with a user interface including prompts, error messages etc.

C.3 Final exercise: a navigational database

This final exercise will be concerned with the design and implementation of a small navigational database.

Its possibilities are so limited and consequently its structure is so simple that it might perhaps better be called a "databox". Yet it shows a number of realistic traits and is suitable for generalization to a quite useful system.

C.3.1 Description of the system

The semantic universe we are talking about consists of *entities*, between which certain *relations* hold. We will introduce commands to establish relations and to delete them. In this process, entities may be added.

This semantic universe can be modeled as a directed labeled ordered graph, in which the entities are represented by the nodes (each node having a unique name, so that nodes with the same name must be the same node) and relations are represented by directed arcs (each arc having a name.)

Relations will be left-unique, i.e. all outgoing arcs of a node will differ in name, and arcs with the same name going out from one node must be the same arc. Names will be of form

$$letter\ \{\ letter \mid digit \mid -\ \}^*$$

using the dash as a visible space within the names.

Initially, the graph consists solely of one node, the *starting node* of the graph, labeled `Start node`.

A session is a dialogue, consisting of commands in one direction (input) and information displays in the other direction (output), in which the graph can be examined and modified. From one session to the next the graph may be preserved as a data file.

C.3.2 Navigation

We wish to provide commands for navigating over the graph structure of the database, following arcs by giving their name and in this way traversing various nodes. In this navigation it is very important that the user should at all points know where she is. Therefore we will, after each command, prompt the user by giving the name of the current node. The farther we are away from the starting node, the deeper we will indent this node, in this way emphasising the following of an arc and the return to a previous node.

Initially, the current node is the starting node and the indentation will be, e.g. twenty positions.

C.3.3 The data manipulation language

We will start by describing an interactive data manipulation language for examining and modifying the database. The *commands* of the database manipulation language will be in free format style (that is, spaces will serve to separate the elements of a command), with one command per line. We will distinguish the following commands, which all consist of a single character, possibly followed by one or two names:

```
+ selector target
```

The *add-command* consists of a plus-sign followed by two names, the name of the selector and the name of the target. It adds the relation `(current node, selector, target)` to the contents of the database, provided the current node does not yet have an outgoing arc named `selector`. In case the target name is one which has not appeared before, it is added to the universe of entities.

```
=
```

The *list-command* consists of a single equal-sign. It lists the names of all outgoing arcs of the current node (all “selectors”).

```
> selector
```

The *follow-command* consists of a greater-sign followed by the name of the selector to be followed. If the current node has an outgoing arc of the name `selector` to some target node, the indentation is increased, by e.g. four spaces, and that target node is made the current node.

```
<
```

The *return-command* consists of a less-sign. It makes the previous node, i.e. the node from which we reached the current node, the current node and decreases the indentation. If there is no previous node the dialogue halts after writing the results of our work into a file and displaying the current storage utilization.

C.3.4 Example session

In order to illustrate both the intended user interface and the commands we will give an example session. The program identifies itself and then ask whether we want to load an existing database possibly saved in a previous session. Here, we decide not to load a database and the program inquires the name of the starting node.

```
Info System
         Start node
?
```

We type the name of the start node

```
? William
```

From now on, whenever the program wants to read input, it displays the name of the current node and prompts with a question mark on a new line for further input, e.g.

```
                    William
?
```

Just now the current node is still the start node. We now add an arc to this node by the command

```
? + wife Mary
```

The system responds with

```
                    William
```

and we continue with adding a second relation

```
? + daughter Julia
                    William
```

Note that the current node still is our start node. We have now obtained the following information in the database:

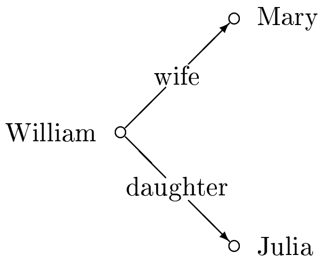


Figure C.1: Situation after two add commands

We give the list-command in order to see what relations we have established and obtain:

```
? =
            wife
            daughter
          William
```

Following the daughter-relation we change the current node

```
? > daughter
            Julia
```

and continue adding two relations

```
? + husband John
            Julia
? + daughter Cassandra
            Julia
```

If we attempt to add a second relation named `daughter`, this will be reported as an error, e.g.

```
? + daughter Alexandra
Selector not unique, ignored
            Julia
```

But we can add

```
? + father William
            Julia
```

If we follow the father-relation

```
? > father
```

we arrive again at the node `William`, but now at a deeper level.

```
              William
```

We return by successive commands to the root

```
? <
            Julia
? <
          William
```

At the root, an additional return-command brings us out of the program

```
? <
Storage for text= 46
Number of work cells left= 98
```

reporting the amount of storage used by the added relations and writing all information entered so far to a file.

C.3.5 Reading and saving the database

The database obviously needs a mechanism for storing the results of a session until the start of a next session. It would of course be possible to keep and update the contents of the database on a file, but is also sufficient to keep it in memory during a session but to read it from file at the start of the session and write it to a file at the end of the session.

Of course the contents of the database could be dumped in some numeric form but the following scheme is more attractive. At the end of the session, the current contents of the database is written onto a file in the form of a sequence of commands which, when executed, reconstruct the current contents of the database.

Programming this is a nice exercise in recursion. For this approach to work, the input strategy must be such that input takes place initially from a file and changes to the keyboard upon meeting the end of that file.

The technique described is useful in many different situations and surprisingly easy to implement.

C.3.6 Possible extensions

It is not hard to suggest a number of sensible extensions to the database just described that make it a lot more interesting and possibly even useful.

Obviously a delete command is missing, with a syntax like

```
- selector
```

to delete the relationship with that selector from the current node. It is not difficult to add, but it is not immediately clear what to do if, through the deletion of a relation, some entity becomes unreachable. The easiest solution is to simply leave the entity in the text table, since there is no point in explicitly removing it.

Similarly, it might be useful to add a command for jumping immediately to a node with a specific name, e.g. in the form

```
! name
```

meaning: jump immediately to the node of that name (note that, because of the reading strategy followed, once the command has been read a node of that name exists, even if there is no arc leading to or from it.)

Further extensions are of course possible, bringing the database nearer to a real life system:

- it is possible to impose on each node a type, like integer, real, text or table (with a certain structure), so that different forms of information can be kept in the database.
- it is possible to impose an a-priori structure on the database by fixing the number of fields and the names of the selectors a node can have. One attractive technique is to equip the database with a description of its contents in the form of a context-free grammar.

- we can also generalize this database for simultaneous access by many people, using some form of lockouts for safe simultaneous update. And we can distribute it over many computers. The sky is the limit.

The small database described is a good starting point for experiments in the construction of all kinds of software, not only databases — The reader is invited to exercise her own phantasy.

Bibliography

[ASU86] A. V. Aho, R. Sethi, J. D. Ullman, *COMPILERS — Principles, Techniques and Tools*, Addison-Wesley publ. co., 1986

[ALL70] F. E. Allen, *Control Flow Analysis*, SIGPLAN Notices 5/7, July 1970

[BAY81] M. Bayer, B. Böhringer, J.-P. Dehottay, H. Feuerhahn, J. Jasper, C. H. A. Koster, U. Schmiedecke, *Software Development in the CDL2 Laboratory*, in: [HUN81]

[COU77] P. Cousot, R. Cousot, *Static Verification of Dynamic Type Properties of Variables*, in: Proceedings 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, St. Monica, CA, Jan. 1977

[DRE75] F. DeRemer and H. Kron: *Programming-in-the-Large versus Programming-in-the-Small*; Proceedings of the International Conference on Reliable Software; SIGPLAN Notices Vol. 10 No. 6, June 1975

[FEU78] H. Feuerhahn, C. H. A. Koster, Static Semantic Checks in an Open-Ended Language; in: P. G. Hibbard and S. A. Schuman (Eds.) *Constructing Quality Software*; North-Holland Publishing Co., 1978

[FOS76] L. D. Fosdick, L. J. Osterweil, *Data Flow Analysis in Software Reliability*, ACM Computing Surveys 8/3, Sept. 1976

[GRU82] D. Grune, *On the design of ALEPH*, CWI Tract 13, Centre for Mathematics and Computer Science, Amsterdam 1982

[HUN81] Horst Hünke, (ed.): *Software Engineering Environments*; North-Holland Publishing Co., 1981

[KIL73] G. A. Killdal, *A Unified Approach to Global Program Optimization*, ACM Symposium on Principles of Programming Languages, Boston, October 1973

[KOS71b] C. H. A. Koster, *Affix Grammars*, in: J. E. L. Peck(Ed.), ALGOL 68 Implementation, North-Holland Publishing Co., Amsterdam 1971

[KOS74b] Cornelis H. A. Koster, Using the CDL Compiler Compiler, in F. L. Bauer(Ed.), J. Eickel (Eds.), *Compiler Construction, An Advanced Course*, Lecture Notes in Computer Science 21, Springer Verlag, Berlin 1976

[KB91] Cornelis H.A. Koster and J. Beney (1991), On the Borderline between Grammars and Programs. In: Proceedings PLILP'91, Passau, Springer Lecture Notes in Computer Science 528, p. 219-230.

[LAM77] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchel, and G. L. Popek, *Report On The Programming Language Euclid*, SIGPLAN Notices vol 12, no 2, February 1977

[LAN73] H. Langmaack, *On Correct procedure parameter transmission in higher programming Languages*, Acta Informatica 2, 1973

[STA80] H. M. Stahl, *Portability and Efficiency in Using an Open-Ended Language*, in Angewandte Informatik 1980

Index