Institute of Informatics
Uniwersity of Warsaw

Report on the Loglan 82 Programming Language

by

Wiesława Maria Bartol, Paweł Gburzyński, Piotr Findeisen, Antoni Kreczmar, Marek Lao, Andrzej I. Litwiniuk, Tomasz Müldner, Wojciech Nykowski, Hanna Oktaba, Danuta Szczepańska, Andrzej Salwicki

---

edited by A. Salwicki 1993, LITA Pau, 2013 Chair of Informatics, UKSW, with contributions of Bolek Ciesielski and Oskar Świda

[1] http://creativecommons.org/licenses/by-sa/3.0/

Contents

Preface

0.1 Thirty years later

This is a new edition of the report. The project Loglan'82 has more than thirty years and is still alive.

We present the report with the following categories of readers in mind:

- **Ambitious programmers** –Loglan'82 offers a few constructs not known in any other programming language:
  - should you avoid a dangerous phenomenon of dangling references and to manage the memory of objects then use the instruction **kill**(),
  - should you create both concurrent and distributed programming then Loglan'82 offers a uniform model for these two sorts of programming saving you the time of learning,
  - if you wish to learn a genuine protocol of alien call – a truly object mechanism of communication/synchronisation of threads,
  - if you wish to apply a powerful and clean tool of coroutines.
- **Teachers** – Loglan'82 is a good choice if you wish to present all the methods and tools of object programming without passing from one language to another for presenting the complete set of object programming constructs,
- **Researchers** – may find interest in Loglan'82 as it is a product of studies. Many scientific problems were solved before we defined the language and its semantics. Some open problems are still open.

0.2 Why Loglan?

We recommend Loglan'82 to those who are investigating. For the language came out as the result of research on questions like:

- is it possible to inherit from a class in a function?
- how to define inheritance when the extended class is not a brother of the present class?
- is it possible to keep the scheme of accessing to non-local ...
- how to define coroutines in a consistent manner?

- is it possible to define semantics of concurrent and of distributed programming in a uniform way?
- is it possible to deallocate objects ina a safe and efficient way?

We are launching a new project named LEM'12. LEM'12 is going to be yet another object programming language. It will profit from the experience gained by Loglan'82 as well as the other programming languages.

LEM'12 is to be a part of much bigger project SpecVer ...

Chapter 1

Introduction

LOGLAN-82 [1] is a universal programming language designed at the Institute of Informatics, University of Warsaw. The shortest, informal characterization of the language would read as follows. LOGLAN-82 belongs to the Algol family of programming languages. Its syntax, however, is patterned upon Pascal's [5]. Many ideas are borrowed from SIMULA-67 [3]. The language includes also some modern facilities such as concurrency and exception handling.

The characteristic programming constructs and facilities of the language are as follows:

- a convenient set of structured statements,
- block structure, i.e. nesting of modules ,
- procedures and functions,
- classes,
- inheritance, alias prefixing,
- safe, programmed deallocation,
- adjustable arrays,
- formal types and formal procedures,
- coroutines,
- processes, one uniform model, common for concurrent as well as for distributed programming,
- a genuine protocol of alien call of methods of one process' object from another process,
- encapsulation techniques,
- exception handling,
- file processing.

[1] Much later we learned about another Loglan – an esperanto-like language developed by dr C. Brown in US.

LOGLAN-82 history

In the early seventies the Institute of Mathematical Machines "MERA" (with two members of the present team of authors) and the Institute of Informatics of Warsaw University initiated the design of a new high level programming language[2]. There were two main inspirations for taking up this research. First, the awareness that the SIMULA 67 programming language was a substantial contribution to the software methodology, and second, that the fast development of multiprocessor hardware will change the software practice. We began our work with analytical studies, seminars and lectures dealing with the basic constructs and features of the known programming languages. This helped us to establish the goals a new programming language should reach. By then, however, we decided that the design of the programming language would be a component of a broader software project, called LOGLAN.

There is no doubt that the environment in which our investigations have been carried out has shed a new light on these goals. In particular, the experience accumulated by a big part of our team in the field of Algorithmic Logic [4][15] influenced the form of the solutions accepted.

The first step of our work was finished in 1977 with the report on the LOGLAN programming language [6][12]. The report provided a general outline of a universal programming language. Among its most important features let us mention a new approach to arrays, assignments, parameter transmission and parallel computations. This version was not implemented. It constituted the base for the agreement between the University of Warsaw and the State Industrial Trust MERA, signed a year later.

A careful analysis of the constructs suggested in the primary project preceded an actual implementation. With the intention of attaining this, the interpreter of the language was designed. At that stage a number of important modifications were introduced to the proposed outline. They resulted from experiments with the interpreter which proved the usefulness of some constructs and the uselessness of some others. At the next stage of research the language was implemented on the original Polish two-processor minicomputer MERA 400. The design was restricted in several points because of the implementation constraints. Some constructs were rejected, the decision concerning some others was put off until a more elaborate analysis was carried out. The experience of the team in the field of abstract data types and computational complexity helped us to solve one of the most fundamental implementation problems - a proper structure for secure and fast storage management. In consequence, the language is furnished with a programmed deallocator which allows the user to design the best strategy of storage management at run time. The implementation of unrestricted prefixing needed a completely new approach. The well-known mechanisms like Dijkstra's display do not allow us to release the SIMULA restrictions (the most important forbids the use prefixing at different levels of unit nesting). Such a solution was found and the LOGLAN-82 users may apply prefixing at an arbitrary level of unit nesting.

Of the results we have obtained so far let us mention paper [2][1], which deals with the principles of an efficient implementation of programming languages with prefixing at many levels. The paper introduces the generalized

[2] words written in 1984

display mechanism and proves the correctness of an update-display algorithm. A new data structure for efficient and secure storage management is also provided. Paper [1] deals with the design and implementation of class Simulation (improving that provided in SIMULA 67). The concurency problems are described in the special mathematical model [][19]. The correctness of the monitor implementation is proved in [20]. The semantics of an assignment statement for subscripted variables is defined and carefully examined in [21]. Paper [16] describes the semantics of allocation, deallocation and control statements. A comprehensive survey about LOGLAN-82 and its applications is supplied in [8]. Let us mention the close connections between the development of the language itself and of Algorithmic Logic, see [15, 22, 23, 24, 25, 26].

NEW PUBLICATIONS

NEW WORK

1983 - Summer School on Loglan. Hans Langmaack solves the problem of static binding of identifiers in the presence of multilevel inheritance.
1984 - Pawel Gburzynski and Andrzej Litwiniuk install Loglan on Siemens computer.
1985 - Danuta Wasersztrum-Szczepanska ports Loglan to VAX/VMS system.
J.Findeisen ports to PDP-11.
1986 - Danuta Wasersztrum-Szczepanska ports Loglan to IBM PC.
1987 - continuation of the works on PC
1988 - Bolek Ciesielski proposes and realizes a new concept of parallel processes and a new communication mechanism "alien call". He realizes as well an experimental network of PCs executing Loglan's processes.
1989 A Loglan to C crosscompiler was realized by M.Wojtylak and T.Gottwald.
1990 - J.Bartoszek wrote a structured editor
1991 - Pawel Susicki installs Loglan in Unix environment
1992 - Sebastien Bernard ports Loglan to Atari STE
1993 - distribution of Loglan by network.

LOGLAN-82 high points

- An orderly and intellectually manageable fashion of program design.

- Clean, modular extensibility (by means of the above mentioned facilities, in particular by prefixing). An algorithm employing an abstract data structure can be prefixed by a class realizing that structure. The class may be programmed by the user himself or by another user, taken from the system library etc. In this way, programs may be developed by teams of programmers.

  An environment for distributed and safe development of large programs and systems with easy inter-communication between members of software teams, i.e., different parts of the design are easy to read, check and modify. The modifications do not entail unexpected interactions.

- Possibility of systematic debugging in a way which contributes to confidence in the overall program correctness.

- Type checking, especially of references to objects, which substantially reduces the need for run-time checks and increases the safety of handling pointers.
- Efficient storage management by means of well-tailored allocation/deallocation operations.
- Clear visibility rules with the capability of unit encapsulation techniques.
- Concurrent computations in which several processes are simultaneously and independently executed by any number of processors. The concurrent multiprocessor computations were treated with due care. We reached the necessary foundations for the description of atomic operations for the concurrent statements. The atomic operations may be efficiently implemented in any operating system kernel. It is well known that concurrent computations have to be synchronized and scheduled. We do not prejudge which facilities are to be used for those purposes. In LOGLAN-82 all known synchronization methods may be declared as predefined classes. For example, let us mention that it is possible to define:
- monitoring dialect similar to CONCURRENT PASCAL, cf.[5], with the main notions: process, monitor, entry procedure, delay, continue,
- tasking dialect similar to ADA's tasks, cf.[11], with the main notions: task, accept, select, rendez-vous.

First implementation of LOGLAN-82

The first implementation of the language was finished in December 1981 on the two processors Polish minicomputer MERA-400 (uni-bus architecture). The whole compiler was programmed in FORTRAN IV Standard(!). The run-time system and file processing were coded in the Mera Assembly Language GASS. The implementation team was headed by Antoni Kreczmar (who is the author of Running System) and included Paweł Gburzyński (File Processing), Marek Lao (Semantic Analysis), Andrzej Litwiniuk (Code Generation), Wojtek Nykowski (Parsing) and Danuta Szczepańska-Wasersztrum (Static Semantics).

Further work on LOGLAN-82

Although we are convinced that LOGLAN-82 will prove to be useful for an average user, we would like to stress that we were interested mainly in finding answers to research questions. Our approach is more scientific than commercial. Among the studies that are planned for the nearest future, let us mention further research on LOGLAN-82 itself and on its first compiler. The portability of the compiler seems to be the main target of our team. Moreover, LOGLAN-82 has been used in several applications. In this way the language will be verified and its usefulness will be analyzed. We are convinced that the new computer architecture and multiprocessor environment should be taken into account. Therefore, we plan studies which could support an efficient implementation of the language with richer semantics are planned. It seems that the crucial point of the future hardware would be the efficient implementation of the storage management.

30 years later

[3] From the perspective it is easy to observe two facts:

- Loglan'82 remains forgotten,
- it enjoys several features which are important for software engineering but remain unknown to the community of programmers.

Acknowledgments

We wish to express our gratitude to all institutions and persons who supported us materially or morally. Thanks are due to the State Industrial Trust "MERA" and to its deputy director professor A.Janicki for the arrangements that enabled us to realize the LOGLAN-82 project. The LOGLAN-82 team wishes to thank all colleagues in Warsaw for criticism and helpful remarks. This report has been carefully read by a number of people, including J.Deminet, F.Kluzniak, A.Janicki, J.Rudziński, W.M.Turski. Their critical comments helped us to avoid numerous mistakes.

[3] one

Chapter 2

The basic characteristics of LOGLAN-82

2.1 Control structure

Compound statements in LOGLAN-82 are built up from simple statements (like assignment or call statement) by means of conditional, iteration and case statements.

The syntax of a conditional statement is as follows:

```
if  boolean expression
then
  sequence of statements
else
  sequence of statements
fi
```

The semantics of a conditional statement is standard. The keyword fi allows us to nest conditional statements without the appearence of the "dangling else" ambiguity. The "else" part in a conditional statement may be omitted:

```
if boolean expression
then
  sequence of statements
fi
```

Another version of a conditonal statement has the form:

```
if B1 orif ... orif Bk
then
  sequence of statements
else
  sequence of statements
fi
```

For the execution of a conditional statement with the orif list the specified conditions B1, ..., Bk are evaluated in succession, until the first one evaluates

to true. Then the rest of the sequence is abandoned and the "then" part is executed. If none of the conditions evaluates to true, the "else" part is executed (if any). The orif construction provides a good method for a short circuit technique, since the boolean expression controling the conditional statement execution need not be evaluated till the end.

Similarly, a conditional statement with the andif list has the form:

```
if B1 andif ...andif Bk
then
  sequence of statements
else
  sequence of statements
fi
```

For the execution of this kind of statement the conditions B1, ..., Bk are evaluated in succession until the first one evaluates to false. Then the "else" part is executed (if any). Otherwise the "then" part is executed.

The basic form of an iteration statement in LOGLAN-82 is the following:

```
do
  sequence of statements
od;
```

To terminate the iteration statement one can use the simple control statement exit, which has the following syntactic form:

```
exit  ..... exit
```

repeated an arbitrary number of times. It may occur in a nested loop statement. The execution of exit.....exit (i - times) statement consists in the control transfer to the statement immediately following the i-th od after the exit statement, (where in counting the od's, the pairs do-od are disregarded). In particular, when exit occurs in a simple loop the control is transferred to the statement immediately following the od symbol, which allows us to terminate the loop. Similarly, a double exit terminates two nested loops, a triple exit terminates three nested loops etc. Moreover, a LOGLAN-82 iteration statement allows us to place many loop exit points in arbitrary configurations, e.g., exit may appear in nested conditional statements, case statements, etc.

Iteration statements with controlled variables (for statements) have the forms:

```
for  j := A1  step A2 to  (or downto)  A3
do
  sequence of statements
od;
```

The type of the controlled variable j must be discrete. The value of this variable in the case of the for statement with to is increased, and in the case of the for statement with downto is decreased. The discrete range begins with the value of A1 and changes with the step equal to the value of A2. The execution of the for statement with to terminates when the value of j becomes for the first time greater than A3 (with downto when the value of j becomes for the first

time less than A3). The values of the expressions A1, A2, A3 are evaluated once, upon entry to the iteration statement. The default value of A2 is equal to 1 (when the keyword step and A2 are omitted).

An iteration statement with the while condition has the form:

```
while  boolean expression
do
  sequence of statements
od;
```

and is equivalent to

```
do
  if not boolean expression then exit fi;
  sequence of statements
od;
```

To enhance the users's comfort, the simple statement repeat is provided. It may appear in an iteration statement and causes the current iteration to be finished and the next one to be continued (something like jump to CONTINUE in Fortran's DO statement). In general, this statement has the form:

```
exit ... exit repeat
```

and causes the current iteration of the corresponding enclosing iteration statement to be finished and the next one to be continued.

A case statement in LOGLAN-82 has the form:

```
case A
  when Q1 :  G1
  when Q2 :  G2
     ...
  when Qk :  Gk
  others      G
esac
```

where A is an arithmetic expression, Q1, ..., Qk are constants and G1, ..., Gk are sequences of statements. A case statement selects for execution a sequence Gj where the value of A equals Qj. The choice others covers all values (possibly none) not given in the previous choices.

2.2 Block structure

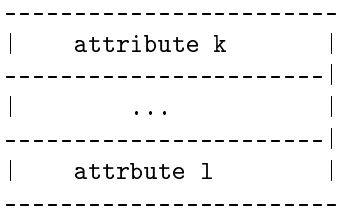
LOGLAN-82 adopts and extends the main semantic features of the ALGOL family programming languages (ALGOL-60, ALGOL-68, SIMULA-67) i.e., the block structure. The block concept of ALGOL-60 is a fundamental example of this mechanism. The syntactic structure of a block is as follows:

```
block
  list of declarations
begin
  sequence of statements
end
```

The list of declarations defines some syntactic entities, e.g. constants, variables, procedures, functions etc., whose scope is that block. The syntactic entities occurring in the sequence of statements are identified by means of identifiers which are introduced in the declaration lists. For every identifier occurrence it must be possible to identify the corresponding syntactic entity. This kind of correspondence between occurrences of identifiers and syntactic entities is necessary to define the semantics of a block statement. The block statement semantics may be described as follows.

When a block is entered, a dynamic instance of the block is generated. In a computer, a block instance takes the form of a memory frame containing syntactic entities declared in that block. All local syntactic entities of an instance will be called its attributes .

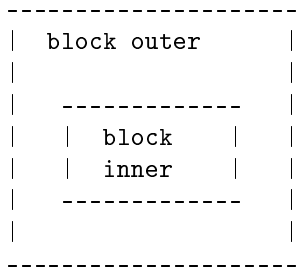
The frame of a block instance may be viewed as a box (with displayed attributes when necessary).



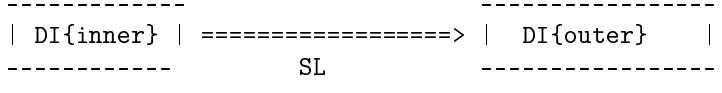
a block instance

A block is a statement, and so other blocks may occur in its sequence of statement (i.e., blocks may be nested). Observe, that the occurrences of identifiers in an inner block need not be local. They can refer to entities declared in the outer block. For a non-local occurrence of identifier, the corresponding attribute of a non-local instance should be identified. That identification is possible thanks to an auxiliary notion of a syntactic father.

Consider the following block structure:



When the statements of block2 are executed, the following two dynamic block instances are created:



Here O[1] is an instance of the block1, and O[2] is an instance of the block2.

The instance O[1] is called the syntactic father of O[2] (or alternatively the instance O[2] is syntactically linked by the SL-link with the instance O[1]). During a program's execution the sequence of syntactic fathers determined by an active instance forms a chain, called an SL-chain. The instances forming the SL-chain correspond to the consequtive enclosing units of the program, starting from the active one and ending on the main block. Thus, this chain allows us to identify all non-local occurrences of identifiers.

A block statement terminates when the control reaches its final end, and then its instance is automatically deallocated.

2.3 Procedures and functions

A block statement is the simplest example of a unit. Upon execution of a block statement an activation record is generated. The instructions of the block are executed in the environment modelled upon the declarations contained in the block. The activation record is deallocated automatically when the end symbol is reached. Procedures and functions constitute the next step of know-how in high level programming languages.

The syntactic form of a procedure declaration is as follows:

```
unit name: procedure(formal parameters);
  list of declarations
begin
  sequence of statements
end;
```

A procedure is a named syntactic unit which may be invoked only via its identifier by means of a call statement:

call name (actual parameters);

(Procedures differ from blocks also in that they can have parameters, but this question will be discussed later.)

When a procedure is called, its instance is created, as in the case of a block. All local attributes are allocated in the new frame. A syntactic father of such a newly generated instance is defined as usual, and allows us to identify all non-local attributes.

A procedure call is terminated when the control reaches return statement or the final end. Then the control returns to the instance where the procedure was called. That instance is referred to by another system pointer (DL-link).

After the termination of a procedure call there is no syntactic means to access its local attributes, hence its instance is automatically deallocated.

Functions differ from procedures only in that they return a value and are invoked in the expressions.

2.4 Classes

To meet the need for permanent data structures LOGLAN-82 introduces the notion of class (cf [3]). Class is declared in a similar way to procedure. It is named and may have parameters:

```
unit M :class(formal parameters);
  list of declarations
begin
  sequence of statements
end M;
```

The main difference between classes and procedures consists in the way the instances of these syntactic units are treated. (To distiguish class instances from those of blocks, functions and procedures they will be called class objects or simply objects). The class generation yields a class object which is a permanent data unlike the vanishing procedure (function, block) instance. The object O of class M is generated by the object generator statement:

```
new M(actual parameters)
```

This statement invokes the same sequence of actions as a procedure call, i.e., it opens a new object, transmits parameters and executes the sequence of statements of M. Return to the caller is made by the execution of a return statement or when the final end is reached. The access to such an object is then possible if its address is set to a variable. The variables whose values point to class objects are called reference variables. A reference variable of type M is declared as follows:

```
var X: M;
```

and may point to any object of class M, for instance, the statement:

```
X:=new M(...)
```

generates an object O of class M and assigns its address (reference) to X. The default value of any reference variable is none, which denotes fictitious non-existing object. What is left behind is a structure of attributes which can be accessed by means of dot-notation. These accessible attributes are either formal parameters or local entities. If X is a reference variable of type M and W is an attribute of class M, then the remote access to the attribute W has the form:

```
X.W
```

The above remote access is correct if X points to an object O of class M. Otherwise a run time error is raised (for instance when the value of X is none).

2.5 Inheritance alias Prefixing

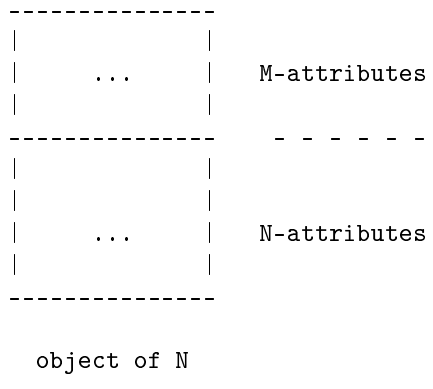
Prefixing is another important programming facility borrowed from SIMULA-67. Its most important feature consists in the possibility of unit extension. Consider the following example. Let M be a class:

```
unit M:  class;
  list of declarations of M
begin
  sequence of statements of M
end ;
```

Now let N be a class:

```
unit N: M  class
  list of declarations of N
begin
  sequence of statements of N
end ;
```

Class N is prefixed by class M. The name of the prefix is located immediately before the symbol class. Class N is treated as an extension of M, i.e., the object of class N has a compact frame consisting of the attributes of N as well as the attributes of M:



The structure of such an object is determined by the class M as well as by N (thus containing both M-attributes and N-attributes). The statement

```
X:=new N    ,
```

where X is a variable of type N, creates an object of class N.

The sequences of statements of classes M and N are also concatenated. In the sequence of statements of a class the keyword inner may occur anywhere, but once only. The sequence of statements of N consists of the sequence of statements of M with inner replaced by the sequence of statements of N (inner in N is equivalent to an empty statement). If class N prefixes another class P, then inner in N is replaced by the sequence of statements of P, and so on. If inner does not occur explicitly, an implicit occurrence of inner just before the final end of class is assumed.

Prefixing allows the programmer to extend units. Assume, for instance, that STACK is the data structure which defines a push-down memory:

```
unit STACK :class;
   ...
  unit pop: function...
  end;
   ...
  unit push: procedure...
  end;
   ...
```

```
begin
    ...
end STACK;
```

Any class prefixed by STACK inherits the operations on stack. For instance, in a class declaration

```
unit N:  STACK class;
     ...
  begin
     ...
     call push;
     ...
  end ;
```

the function pop and the procedure push may be used as any other local attribute.

A class may also be used to prefix blocks, procedures and functions. An instance of a prefixed block is a compound object and is created upon entry to the block and deallocated after its termination, as in the case of a simple block. Similarly, an instance of a prefixed procedure (function) is a compound object which is created when a procedure (function) is called and deallocated after its termination.

2.6 Object deallocator

The classical methods used to deallocate class objects are based on reference counters or garbage collection. Sometimes both methods may be combined. The reference counter is a system attribute holding the number of references pointing to the given object. Hence any change of the value of a reference variable X is followed by a corresponding increase or decrease of the value of its reference counter. When the reference counter becomes equal to 0, the object can be deallocated.

The deallocation of class objects may also occur during the process of garbage collection. During this process all unreferenced objects are found and removed (while memory may be compactified). In order to keep the garbage collector able to collect all the garbage, the user should clear all reference variables, i.e., set to none, whenever possible. This system has many disadvantages. First of all, the programmer is forced to clear all reference variables, even those which are of auxiliary character. Moreover, the garbage collector is a very expensive mechanism and thus can be used only in emergency cases.

In LOGLAN-82 a dual operation to the object generator, the so-called object deallocator is provided. Its syntactic form is as follows:

```
kill(X)
```

where X is a reference expression. If the value of X points to no object (none) then kill(X) is equivalent to an empty statement. If the value of X points to an object O, then after the execution of kill(X) the object O is deallocated. Moreover, all reference variables which pointed to O are set to none., This

deallocator provides full security, i.e., the attempt to access the deallocated object O is checked and results in a run-time error. For example,

```
Y:=X;  kill(X);   Y.W:=Z;
```

causes the same run-time error as

```
X:=none;  X.W:=Z;
```

The system of storage management is arranged in such a way that the frames of killed objects may be immediately reused without the necessity of calling the garbage collector, i.e., the relocation is performed automatically.

2.7 Arrays

LOGLAN-82's array is a kind of a class with indices instead of identifiers selecting the attributes. A variable of an array type is a reference variable pointing to an object which contains components of a one-dimensional array. The components of such an array may also point to one-dimensional arrays and so forth, hence multi-dimensional arrays may be generated as well.

The declaration of a variable Y of array type has the following form:

```
var Y :  arrayof  ...  arrayof  T
```

where the number of arrayof defines the dimension of Y. The declaration of a variable Y fixes its dimension, while the bound pairs are still undetermined. The array generation statement has the form

```
array  Y  dim  (l : u)
```

where l, u are arithmetic expressions determining the lower and upper bounds of the first index. The object O of an array is generated and the reference to O is assigned to Y.

If Y is declared as a two-dimensional array, then one can generate a two-dimensional array by means of the statements

```
array Y dim (l:u);

for i:=l to u
do
  array Y(i) dim (li:ui)
od;
```

where the shape of each row is determined by the bounds li, ui. Hence triangular, tridiagonal, streaked arrays, etc. may be generated. Moreover, the assignment statements allow us to interchange array references that are of the same dimension and the same type, e.g. Y(i):=Y(j). In consequence, the user may operate on array slices. The default value of any array variable is none, as in the case of a reference variable.

2.8 Parameters

In LOGLAN-82 there are four categories of parameters: variable parameters, procedure parameters, function parameters and type parameters.

2.8.1 Variable parameters

Variable parameter transmission is simplified in comparison with ALGOL-60 and SIMULA-67. There are three transmission modes of variable parameters: input mode, output mode and inout mode. In the syntactic unit which is a procedure, a function or a class, the formal input parameters are preceded by the symbol input, the formal output parameters are preceded by the symbol output and the formal inout parameters are preceded by the symbol inout. The default transmission mode is input. Input parameters are treated as local variables initialized by the values of the corresponding actual ones. Output parameters are treated as local variables initialized in the standard manner (real with 0.0, integer with 0, reference with none, etc.). Upon return their values are assigned to the corresponding actual parameters, which in this case must be the variables. Inout parameters act as input and output parameters together.

2.8.2 Procedure and function parameters

In LOGLAN-82 procedures and functions may also be formal parameters. This category of parameters allows us to parametrize a unit with respect to some operations. A formal procedure (function) has the full specification part, i.e., the parameter list (and the function type), for instance :

```
unit Bisec: procedure(function f(x: real): real; a, b, eps:real);
begin
   ...
end;
```

Type parameters Types are also allowed to be transmitted as parameters. This kind of parameters enables us to parametrize a unit with respect to some types. For instance consider the following declaration:

```
unit sort:procedure(type T; A:arrayof T;  function less(x, y:T):boolean);
begin
   ...
end
```

The actual parameter corresponding to the formal T must be a non-primitive type. The array A must be the array of elements of the actual type. If function less defines the ordering relation on the elements of the actual type, then this procedure may be invoked to sort the array A.

2.9 Coroutines

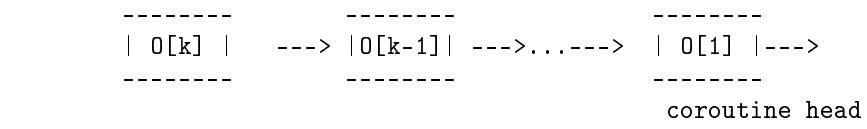
Notion of coroutine is an extension of the class notion. A coroutine object is an object whose sequence of statements can be suspended and reactivated in the programmed manner. The generation of a coroutine object terminates with the execution of the return statement (then the control is passed to the caller as in the case of classes). A coroutine object after the execution of the return statement is suspended. A suspended coroutine object may be reactivated with the help of the attach statement:

```
attach(X)
```

where X is a reference variable designating the activating object.

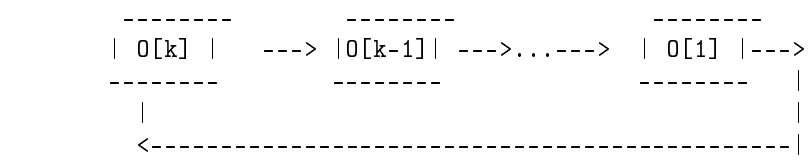
In general, from the moment of generation a coroutine object is either active or suspended. Any reactivation of a suspended coroutine object O causes the active coroutine object to be suspended and continues the execution of O from the statement following the last executed one.

During a coroutine execution some other unit instances may be generated. They are dynamically dependent on that coroutine object. Thus, an active coroutine (in particular the main program) can be illustrated by the following chain:



where the arrows denote the DL-links.

This DL-chain is transformed into the DL-cycle when the control is transferred to another coroutine as the result of the attach statement.



2.10 Processes

The concept of process in LOGLAN-82 is a natural extension of coroutine. Coroutines are units which once generated may operate independently, each one treated as a separate process. For coroutines, however, an essential assumption is established; namely, when one coroutine object is activated, the active one must be suspended. When processes are used, the activation of a new process does not require the active one to be suspended. Thus during a program's execution many processes may be active simultaneously. Their statements are computed in parallel. There are two operations, stop and resume, which concern the control of processes.

stop Operation which causes the active process to be stopped. resume(X) Operation which reactivates the process referenced by X.

Synchronization and scheduling.

Elementary synchronization in LOGLAN-82 is achieved by two-valued semaphores and a number of simple indivisible statements operating on them. These statements are the following (where Z denotes a variable of semaphore type):

ts(Z) Test-and-set boolean function which closes semaphore Z and returns the value true if Z was open and false if Z was closed. lock(Z) Operation which tests the value of the semaphore Z and either enables the given process to enter the critical region guarded by Z (if Z is open) or suspends the process (in the opposite case) until another one opens that critical region. unlock(Z) Operation the execution of which opens the critical region guarded by Z. stop(Z) Operation that opens the critical region guarded by Z and stops the execution of the given process.

The above operations are implemented in the kernel of the operating system. One can use them to define any complex synchronization facility, e.g., monitors (cf. 11.3.). Once defined and stored in the library, the facility can be used by any user. Moreover, using the high level synchronizing tools, the user can cover the low level, primitive ones (therefore the properties of high level tools cannot be disturbed). There is also a parameterless function wait. If wait is called in the given process X, then process X waits for the termination of any of its son (a son of X is a process which was generated in X). The returned value of wait points to the first terminated son, and then, the computation of process X is continued. If there is no such son, the returned value of wait is none.

2.11 Other important features

In LOGLAN-82 the access control mechanism is enlarged so that it supports the data encapsulation technique and the protection of attributes in different environments. The mode of accessibility to attributes of a class can be controlled by means of the specification hidden and close. On the other hand, the mode of accessibility to attributes of a unit that are inherited from its prefix can be controlled by means of the specififfication taken. This permits more flexible communication across the unit boundary as well as defining of abstract behaviour with a hidden auxiliary structure. (For details see 6).

The language provides facilities for dealing with run time errors and other exceptional situations raised by the user. These events are called exceptions. So, the exceptions cause interruption of a normal program execution. The response to an exception is defined by an exception handler. The user is allowed to define the actions that should be raised when an exception is encountered. (For details see 10).

Program units can be compiled separately. Two kinds of separately compiled units are provided: binary items ready to be executed, and library items. The purposes of separate compilation are the following: creating user libraries, handling system and user libraries, compiling program components during program testing, and program overlaying. (For details see 12).

Input-output facilities and file processing are defined by means of some simple primitives. The user is able, however, to declare in the language any class that provides high-level and secure file operations. Examples of system classes

that deal with high-level file operations are also given. (For details see 13).

Chapter 3

Lexical and textual structure

The basic character set consists of

(a) 26 upper case letters:

```
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
```

(b) 10 digits:

```
0 1 2 3 4 5 6 7 8 9
```

(c) 16 auxiliary characters:

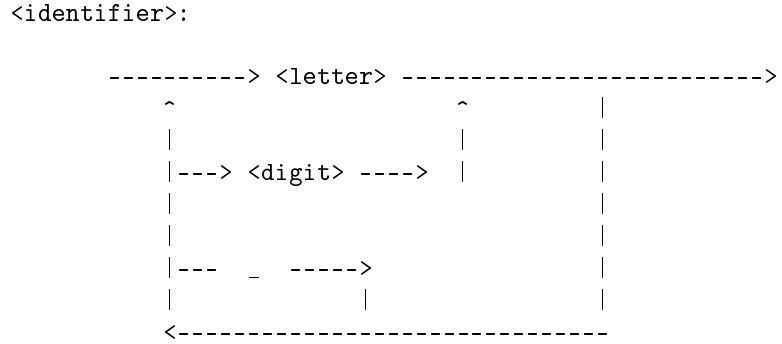
```
. : , ; _ = / + - * < > ' " ( )
```

(d) the space character
This set can be extended with the following characters:
(e) lower case letters
(f) other special ASCII characters, e.g.:

```
# $ ?  % ^
```

(lower case letters are equivalent to the corresponding upper case ones.)

A finite sequence of characters is called a word. The words called identifiers have a special meaning. They are composed of letters, digits, and underscores and start with a letter:



Identifiers serve to identify program entities, i.e., constants, variables, types, functions, procedures, classes, coroutines and processes. There are a certain number of predefined system identifiers which have special significance in the language. The following system identifiers are reserved words (these identifiers cannot be declared by the programmer).

and_if	do	input	others	taken
and	downto	inout	output	terminate
array		is		then
arrayof	else		pref	this
attach	end	kill	procedure	to
	esac		process	type
begin	exit	lastwill	put	
block				unit
	fi	main	qua	
call	for	mod		
case	function		raise	var
class		new	read	virtual
const	get	none	readln	
copy	handlers	not	repeat	
coroutine	hidden		return	wind
		od		when
detach	if	open	signal	while
dim	in	or	step	write
div	inner	or_if	stop	writeln

The lexical entities are identifiers, numbers, strings and delimiters. The delimiters from the basic character set are:

```
, ;  = / + - * > < . ( ) :
```

and the compound symbols are :

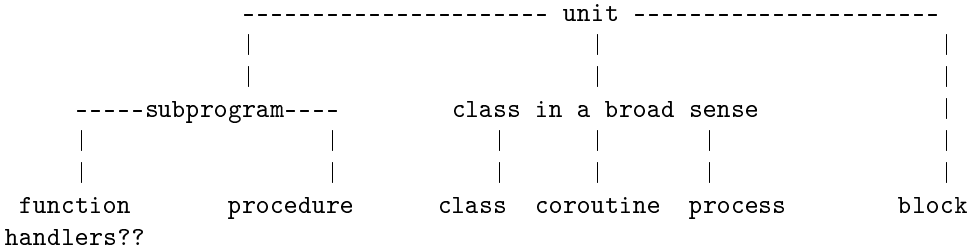
```
=/=   >=   <=  :=
```

Spaces play the role of separators, i.e., at least one space must separate adjacent identifiers or numbers. The end of each line is equivalent to a space.

A comment starts with a left parenthesis and an asterisk and is terminated by an asterisk and a right parenthesis. It may only appear following a lexical unit or at the beginning or end of a program entity. Comments have no effect on the meaning of a program and are used solely for program documentation.

By an identifier definition we mean a declaration or description in the list of formal parameters.

The notion of a unit is explained by the following diagram:



Chapter 4

Types

A type T determines a set |T| of values and a family of operations applicable to the elements of the set. Three kinds of types are distinguished: primitive types, system types and compound types. Variables may be declared to be of type T. Depending on the kind of type T we have to distinguish two cases.

a) T is a primitive type. The value assigned to a variable Y of type T must belong to the set |T|.

b) T is a compound or system type. The value assigned to a variable Y of type T must be a reference pointing to an object in the set |T| (for the notion of reference cf 4.3. and 6.3.)

SYNTAX.

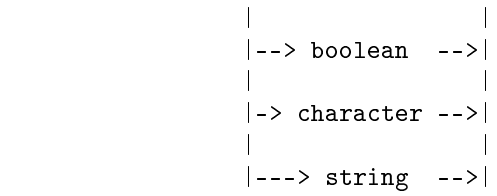
```
          <type identifier>:

      -----> <primitive type> ------>
         |                          ^
         |-> <system type> ------>|
         |                        |
         |-> <compound type> --->|
         |                        |
         |-> <formal type> ------>|
         |                        |
         |-> <file type> -------->|
```

Primitive and system types are pre-defined, compound types are defined by the user. For file type see section 13.

4.1 Primitive types

SYNTAX.

```
  <primitive type>:

      -----> integer  -------->
         |                  ^
         |---> real  ---->|
```



SEMANTICS.

A primitive type determines a finite set of values which can be effectively represented in a computer memory:

|integer| - a subset of integers;

|real| - a subset of reals;

|boolean| - the set consisting of logical values T (true) and F (false);

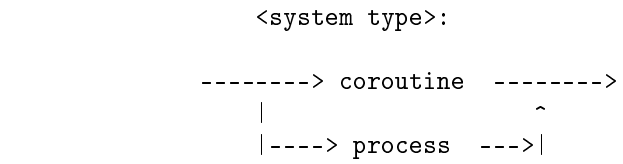
|character| - a set of characters;

|string| - a subset of strings;

These sets will be precisely defined in a concrete implementation. The way in which the primitive type values are represented in a computer memory is not essential for the description of semantics; however, the values of integer and real types are differently represented. Namely, integers are represented in the fixed-point form with a point after the last significant digit, reals are represented in the floating-point form. So they will be denoted differently, e.g., 2 and 2.0. Those values can be mutually converted: the value of type integer is converted to type real by means of conversion into the floating point form; the conversion into the opposite direction truncates and transforms the real value into the fixed-point form.

4.2 System types

SYNTAX.



SEMANTICS.

The set |coroutine| is equal to the union of sets |T| for every type T declared as:

- unit T : coroutine

- unit T : process

- unit T : S class

where |S| is already a subset of the set |coroutine|.

The set |process| is equal to the union of sets |T| for every type T declared as:

- unit T : process

- unit T : S class

where |S| is already a subset of the set |process|.

The user may declare a variable of coroutine (process) type, e.g. of the form

```
var X : coroutine;
                              (var X : process;)
```

and then to assign:

```
X:=new T
```

where T belongs to the set |coroutine| (|process|).

The main block belongs to both sets - |coroutine| and |process|. The system variable main gives the reference to the main block. The variable main may occur in the statements attach(main) and resume(main) only.

4.3 Compound types and objects

SYNTAX.

```
<compound type>:

--------> <array type> ---------->
   |                        ^
   |----> <class type>  --->|
```

4.3.1 Array type

Objects of array type will be called array objects or shortly arrays. An array can be considered as a vector; the access to its components is provided by means of indexing.

SYNTAX.

```
<array type>:

------> array_of  -----> <type identifier> ---->
```

SEMANTICS

LOGLAN-82 types can be uniformly denoted in the following way

$$(\texttt{arrayof})^i\texttt{T} = \begin{cases} \underbrace{\text{arrayof ... arrayof } T}_{i \text{ times}}, & \text{for } i > 0 \\ T, & \text{for } i = 0 \end{cases}$$

where T is a type identifier.

For $i > 0$, the set |(arrayof)^{i}T| consists of the array objects. Every array object has the attributes accessed via indices $l, l+1, ..., u$ where l, u are the values of the lower and upper bounds, respectively, and $l \leq u$. The attributes with the indices $l, ..., u$ are of type |(arrayof)$^{i-1}$T|.

Let O be an arbitrary fixed array object and let Y be a variable whose value points to O. The operations related to the object O are:

- $Y(j)$, where $l \leq j \leq u$, gives the j-th attribute of the object O,

- lower(Y) and upper(Y), which give the value l and u, respectively.

4.3.2 Class type

SYNTAX.

```
    <class type>:

-----> <class identifier> ----->

    <class identifier>:

------> <identifier> ---------->
```

SEMANTICS

A class T is a description of a data structure consisting of attributes i.e., types, functions, procedures, variables, and a sequence of statements. The family of admissible operations on the objects from the set |T| contains the operations defined in the sequence of statements and those defined in the declarations of functions and procedures. The other operations are related to the notion of remote access. They allow us to operate on the objects of type |T| from outside of them.

4.4 Formal types

SYNTAX.

```
    <formal type>:

-----> <formal type identifier> ----->

    <formal type identifier>:

-----> <identifier> ------------------>
```

SEMANTICS

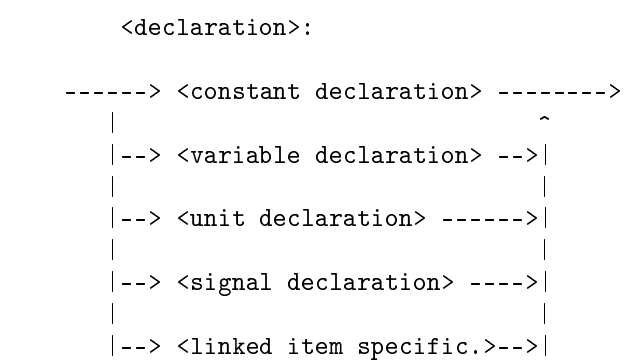
A formal type is a formal parameter of a unit and can be treated as the name of an abstract data structure without any attribute. The corresponding actual type must be a system type or a compound type. The set of objects of the formal type T from a dynamic object O is equal to the set of objects of the actual type which occurs in the actual parameter list of O.

Chapter 5

Declarations

Every identifier which is to be used in a program must be defined. System identifiers are pre-defined, other identifiers are pre-compiled, (see 12.) or they are defined by means of a declaration or description in the formal parameter list. LOGLAN-82 is not strongly typed in the sense that sometimes the type of variable and function value cannot be determined at compilation time. The user may balance the generality and convenience given by the formal types mechanism and the risk of reduced efficiency of his program execution. The compiler option, however, allows us to supress the run time checking with respect to the type compatibility.

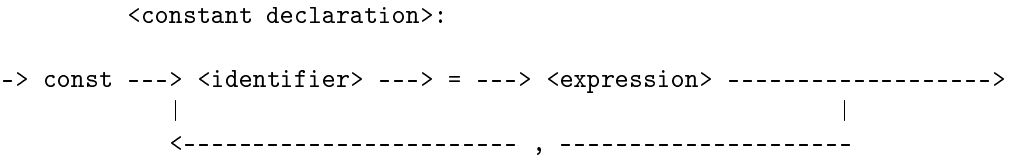
SYNTAX.



(For the definition of a signal declaration see 10.
For the definition of linked item specification see 12.)

5.1 Constant declaration

SYNTAX.



SEMANTICS.

The expression defining the constant must be determinable at compilation time. The type and the value of the constant is given by its declaration. They are always primitive.

Example.

```
const pi=3.1415926, pihalf=pi/2;
```

5.2 Variable declaration

SYNTAX.

```
          <variable declaration>:

---> var ---><specification list>--->


          <specification list>:

----> <identifier list> ---> : ---> <type identifier> ------>
 ^                                                          |
 |<------------------ , <-------------------------------|


  <identifier list>:

-----> <identifier> ------->
   ^                  |
   |<---- , <---------|
```

SEMANTICS.

A variable is of a type given in a variable declaration. A declaration is elaborated at the instant of generation of a unit object which contains that declaration. An elaboration determines an initial value for every variable. This value depends on the type identifier :

```
integer                     -  0
real                          -  0.0
boolean                    -  False
semaphore                   -  open
character and string        -  defined in concrete implementation
any compound and system type -  none
```

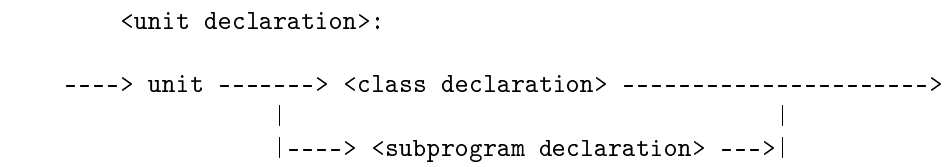
The value of the variable may be modified by means of an assignment statement (see 9.1.1.), but the variables of type T may only point to the object from the set |T|.

Example.

var left, right: node, counter: integer, cycle: arrayof boolean;

5.3 Unit declaration

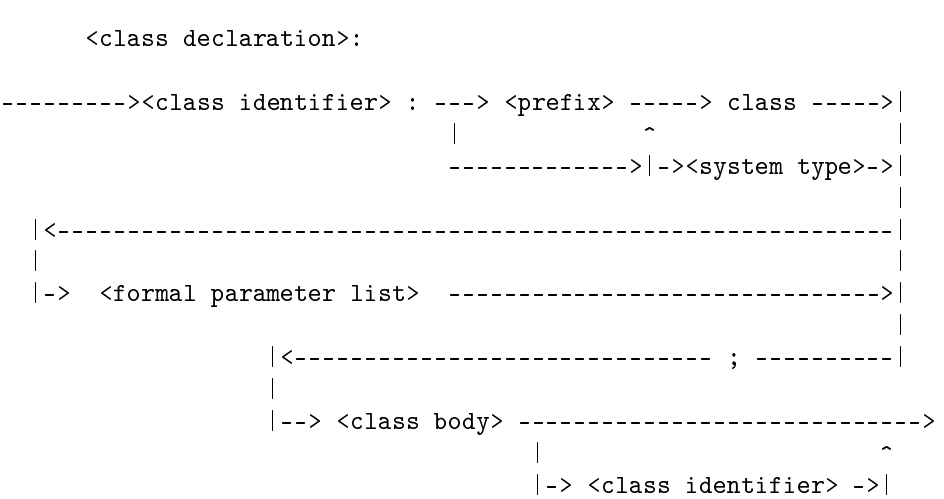
SYNTAX.

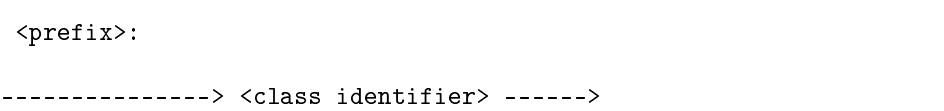


5.3.1 Class declaration (introduction)

A class declaration is understood as a declaration of a class itself, as well as a declaration of a coroutine or a process. The prefixing will be described in 5.3.4..

SYNTAX.



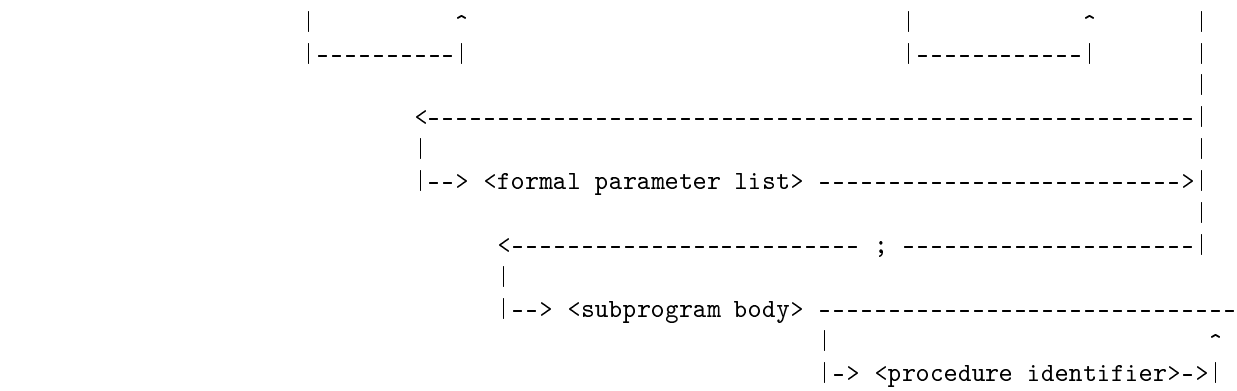


Example.

```
unit complex: class(re, im:real);
  var module:real;
begin
  module:=sqrt(re*re+im*im)
end ;
```

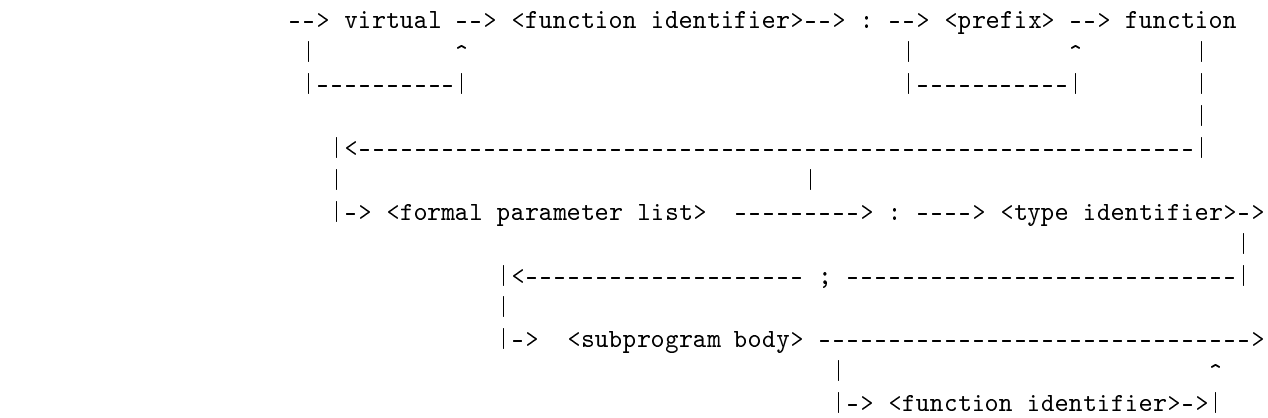
5.3.2 Subprogram declaration (introduction)

SYNTAX.



<procedure identifier> :

---- <identifier> ------->



<function identifier>:

-----> <identifier> ---------->

Class (function, procedure) identifier may optionally follow the end symbol (and if present must match the unit name).

Example.

```
unit Euclid: function(n, m:integer):integer;
   var k:integer;
begin
   do
```

```
      k:=n mod m;
      if k=0 then result:=m; return fi;
      n:=m; m:=k;
    od;
end Euclid;
```

5.3.3 Block

In order to complete the description of LOGLAN-82 units the block syntax is given here, however the occurrence of a block results in the execution of its statements - see 9.1.2..

SYNTAX.

```
     <block>:

 --> pref --> <prefix> ---> <actual parameter list> ---> block ---->
  |                     ^                             ^          |
  |-------------------->|---------------------------->|          |
                                                                  |
                    |<----------------------------------------|
                    |
                    |--> <subprogram body>------>
```

Example.

```
block
  var a, b, c, p, S:real;
begin
  read(a, b, c);
  p:=(a+b+c)/2;
  S:=sqrt(p*(p-a)*(p-b)*(p-c));
  write(S)
end
```

5.3.4 Inheritance or Prefixing

A unit which is a specialized form of a certain class (i.e., which has all the properties of that class and some additional properties defined in the unit) can be defined by means of prefixing. An identifier of the prefixed unit may serve as a prefix for another unit, and so tree structured hierarchies of units can be created. By a prefix sequence of a unit M we mean a sequence M1, ..., Mk of units such that Mk = M, the unit M1 has no prefix; for i = 1, ..., k-1, the unit Mi+1 is prefixed by Mi. Any unit may be prefixed by a class without changing its character (e.g., a prefixed procedure still remains a procedure). Procedures, functions, and blocks cannot be used as prefixes. Process and coroutine, as special cases of class, may also serve as prefixes, but not for procedures, functions or blocks.

If a coroutine (a process) occurs in a prefix sequence of a unit then this unit is treated as a coroutine (a process), and so it has all the properties of a coroutine (a process). Therefore, if a prefix sequence of a unit M contains both

a coroutine and a process then M has the properties of a coroutine as well as those of a process.

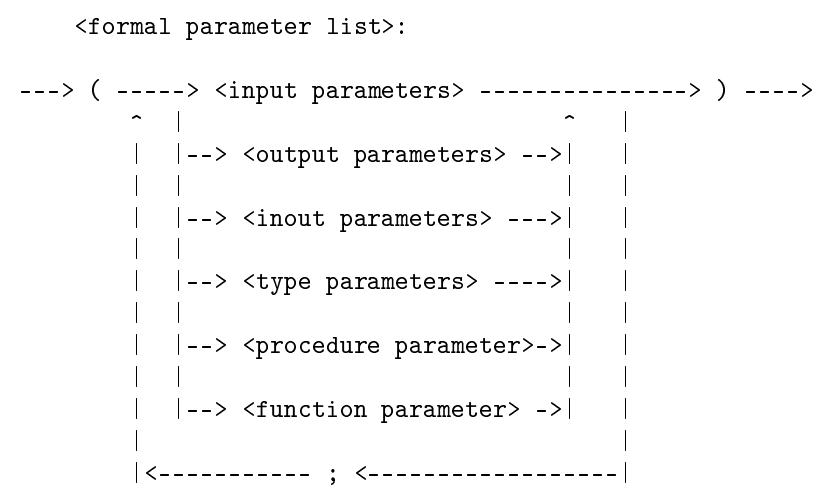
No unit is allowed to occur more than once in its prefix sequence.

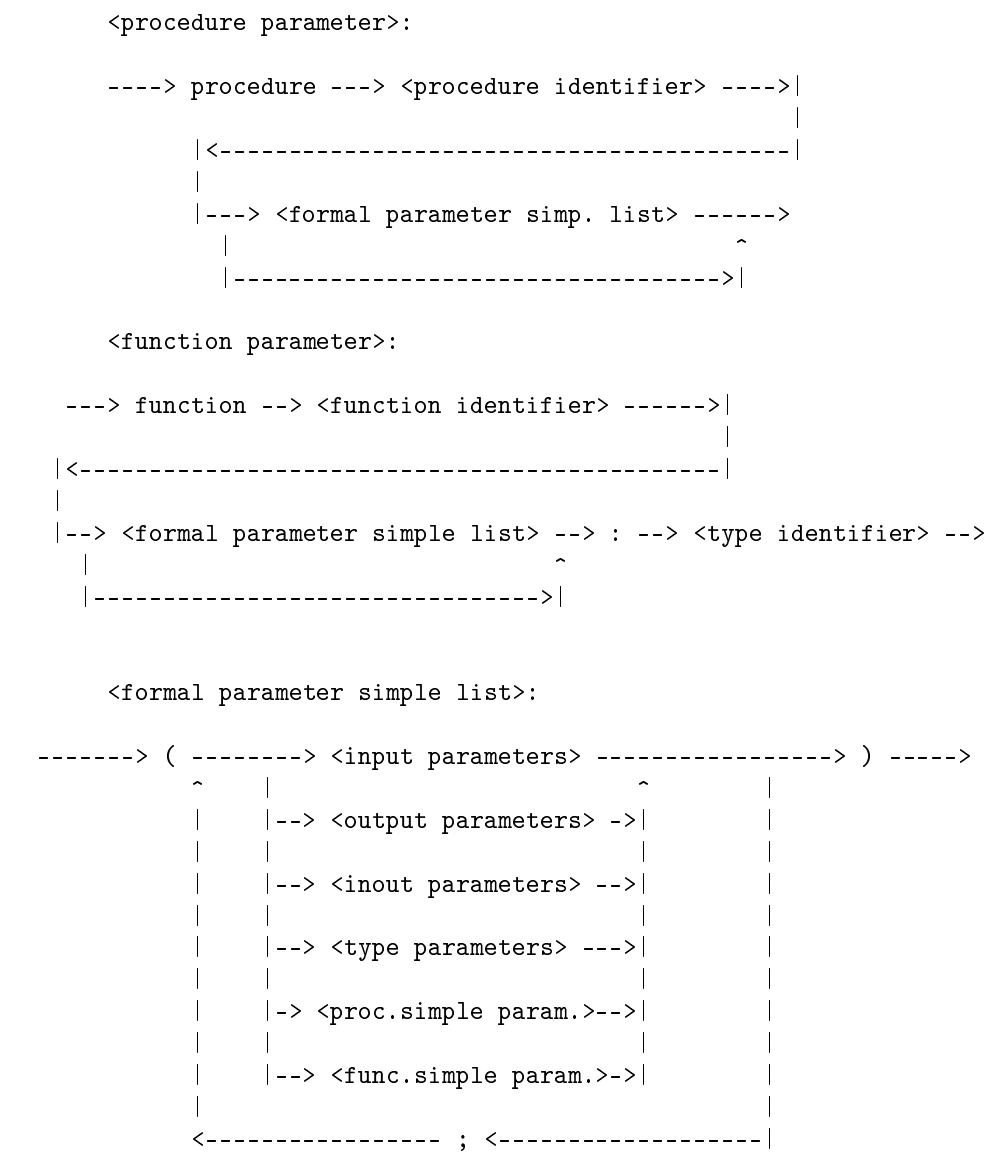
Put T pref* S if a unit T belongs to the prefix sequence of the unit S. Unit S is called a subunit of unit T. As one can see from the definition of object, any object of S has the attributes of the units S and T. Moreover, the statements of that object come from the body of the unit T as well as from that of the unit S.

From the way of implementation it follows that prefixing is not a macro-definition and so it does not require any pre-processing.

5.3.5 Formal parameters

SYNTAX.




<procedure parameter>:
----> procedure ---> <procedure identifier> ---->|
<formal parameter simp. list> ------>
<function parameter>:
---> function --> <function identifier> ------>|
|--> <formal parameter simple list> --> : --> <type identifier> -->
<formal parameter simple list>:
-------> (--------> <input parameters> ----------------->) ----->
|--> <output parameters> ->|
|--> <inout parameters> -->|
|--> <type parameters> --->|
|-> <proc.simple param.>-->|
|--> <func.simple param.>->|
<----------------- ; <-------------------|
<procedure simple parameter>:
----> procedure -----> <procedure identifier> ------>
<function simp. parameter>:
----> function -------> <function identifier> ------>

SEMANTICS.

By a formal parameter list of a unit M we shall mean a concatenated list of formal parameters of the bodies of all units M1, ..., Mk = M from the prefix sequence of unit M (successively from 1 to k). The parameters occurring in a unit declaration are called formal parameters to stress that they constitute a pattern for parameters occurring in the unit body. At the instant of object generation the actual parameters for this generation are fixed. The relations between formal and actual parameters depend on the transmission mode which is specified in the formal parameter list.

Those relations make possible the communication between a unit and its environment. The first mode of transmission rectricts the communication to the input (as the beginning of the body) of the actual parameter value for the corresponding formal parameter. The second mode restricts the communication to the output (as the end of the body) of the formal parameter value for the corresponding actual parameter. The third mode possesses both possibilities of the input and output transmission modes. In all three cases, the formal parameters are considered to be declared in the unit body.

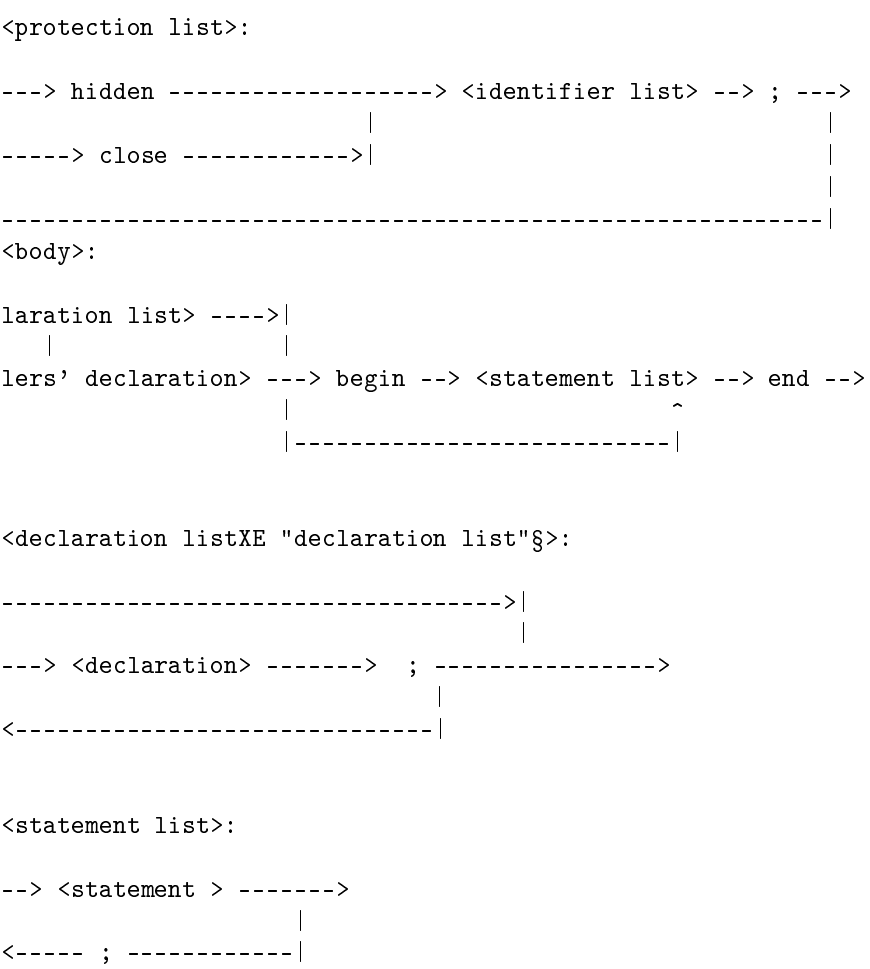
The next two modes of transmission are designed for subprograms and types. The occurrence of a formal subprogram/type in the unit body is matched with the corresponding actual subprogram/type (which is assigned at the beginning of the body execution). In the case of a formal subprogram, a simplified description of its parameters is required.

Hence a LOGLAN-82 unit may be parametrized and designates the union of all units definable by assigning specific actual types to the formal ones. The actual type cannot be a primitive one. Parametrized units make possible the design of universal algorithms, which will be defined in detail at lower levels of program nesting.

5.3.6 Unit body

SYNTAX.

```
          <class body>:

 ---> <inheritance list> ---> <protection list> ---> <body> ----->
  |                      ^ |                    ^
  |--------------------->| |------------------->|


       <subprogram body>:

       ----> <inheritance list> ------> <body> ------>
          |                       ^
          |---------------------->|


       <inheritance list>:

       ----> taken -----> <identifier list> -----> ; ---->
                    |                         ^
                    |-------------------------|
```



SEMANTICS.

In a unit body, a sequence of statements (if any) starts from the begin symbol. Declarations/statements are separated by semicolons. An execution of the unit body begins at the time of the generation of an object (of that unit), see 9.1.2.. A declaration of a unit M is restricted at several points :

Restrictions

(i) The least (textual) unit containing an occurrence of a control statement inner (see 9.1.3.) must be a generalized class. An inner statement may occur in the class body at the most once. If it does not occur explicitly then the body of unit M is assumed to contain the inner statement as the last one (preceding the end symbol).

(ii) All identifiers defined in the body of unit M are different.

(iii) The input/output formal parameters of unit M cannot be of a type declared in unit M.

(iv) If a type T is a formal parameter of unit M then its occurrence in the list of parameters must precede the occurrences of other parameters whose description makes use of T;

Chapter 6

Static and dynamic locations of identifiers. Visibility rules.

As noted before, a non-system identifier used in a program must be defined in the program by a declaration or by a description in a formal parameter list. An identifier need not correspond, however, to only one syntactic entity. A program is composed of units, and so the user designing a unit must pay attention to the relationship between a given unit and the other ones. He should feel free to define his own attributes with the same identifiers as those used in the other units as long as he is not interested in the entities they describe. Therefore some strict rules of correspondence between the identifier and the attribute as well as its valuation are necessary. The first correspondence is called the static location of an identifier, and the second is called the dynamic location. The static location is determined by the syntactic structure of a program. The dynamic location depends on the dynamic configuration of objects. 6.1.

6.1 Unit attributes

A set of attributes is assigned to each unit M. This set consists of all syntactic entities defined in M and in the prefix sequence of M. Most of them form the set of attributes which belong to each object of the unit, i.e., they are dynamic. Virtual functions and procedures are attributes of a very special kind. They are presented separately in 6.4.1. Some other attributes, like constants, are static, i.e., they are not attrributes of the objects of the unit but of the unit itself. Therefore static attributes cannot be accessed by means of dot notation (cf 8.2.3.). The user may protect attributes. The protection mechanisms are introduced in the following sections and discussed in 8.2.3. LOGLAN-82 identifiers cannot be overloaded, i.e., an identifier used in the given unit corresponds to "exactly one" attribute determined by the context. However, identifiers may be redefined. Therefore strict correspondence between the occurrences of the identifiers and the attributes must de defined. Let W be a syntactic entity and M a syntactic unit. We say that W is defined in M iff W is a formal parameter of M (but not of the prefix of M) or W is declared in M. If W is defined in M, the entity it denotes is the meaning of W. From now on we shall use interchangeably the notions of an identifier and of an attribute. Let W be an identifier and M

a unit. If W is defined in M or in a unit from M's prefix sequence, then W corresponds to an attribute of M. More precisely, the corresponding attribute is the one defined in M, if it exists, or the one defined in the prefix sequence. That means that the redefinition of an identifier in a prefixed unit covers the attribute corresponding to that identifier. 6.2. Protected attributes Let us consider a declaration of a prefixed unit. Let M be such a unit and N its prefix. The attributes of N are visible in M (unless covered by the local redefinition). The user, however, can restrict the use of N's attributes in M. The protection may be specified already in unit N as well as in M. The first way corresponds to the hidden specification while the second to the taken specification.

6.1.1 Hidden attributes

A list of hidden attributes is a filter from the prefixing unit. The user may specify within prefix N the attributes whose occurrence is illegal in any unit prefixed by N (unless the identifiers of these attributes are covered by the redeclarations). Remote access to such attributes is forbidden as well (cf 6.2). The absence of hidden specification denotes the empty list. Consider an example:

```
unit N : class;
 hidden x, y, z;
 var x, y, z:integer;
 ...
end N;

unit M:N class;
 hidden x, t;
 var x, y, t:integer;
 ...
end M;
```

Variables x, y declared in N are redeclared in M, and so the identifiers x, y in M refer to the local entities. Variable t is declared in M and is hidden in the units prefixed by M. Variable z is hidden in N, hence it cannot be used in M.

6.1.2 Taken attributes

The list of taken attributes is a filter on the prefixed unit. In unit M the user may specify explicitly the attributes from prefix N which are used in M. Then in M the only attributes accessible from N are those from the taken list. The occurrence of another attribute from N in M's body is illegal. The absence of taken specification denotes the list of all (legal and not hidden) identifiers from N. This means that the user is not interested in the specification of this kind of filtering. The identifiers in the taken list must be defined in the prefix sequence, not in unit M. An exception is an identifier of a virtual attribute (cf 6.4.).

6.1.3 Legal and illegal identifiers

In this section we consider only identifiers corresponding to the attributes of a given unit.

All identifiers defined in a unit are legal in that unit. In particular, all identifiers declared in a non-prefixed unit are legal.

Now let M be a unit, N its prefix and W an identifier not defined in M. Then W is a legal identifier corresponding to an attribute of M iff

- W is legal in N - W does not occur in the hidden list in N - W occurs in the taken list in M or this list is absent

All identifiers specified in every context in a unit must be legal in that unit. All identifiers specified in the taken list must be legal in the prefix.

An identifier is illegal in a unit iff it denotes an attribute of the unit (according to 6.1) and that attribute is not legal.

6.1.4 Close attributes

Close attributes are not accessible by means of remote access (cf. 8.2.3.) outside the unit.

Let M be a unit with the prefix sequence M1, ..., Mk=M. An attribute W of unit M is called a close attribute if W occurs in the close list of Mj for some j, 1<=j<=k, and W is not redefined in any unit following that Mj in the prefix sequence. However, remote access to a close attribute W is allowed within the text of the unit M specifying it to be close, i.e., if the static qualification of the object expression is equal to M, remote access to W is allowed in all the units declared (directly or indirectly) in M.

The list of close attributes must consist of legal identifiers. All hidden attributes are simultaneously close attributes.

Example

```
block
  var v:A;
  unit A: class;
    hidden z;
    close x;
    var x, z:real, y:A;

    unit B:A class;
      var t:B;
      begin

       ... z ...        (* is illegal since hidden in A *)
       ... x ...        (* is legal *)
      .. y.x+y.z ..     (* is legal since y is qualified by A
                           and the expression is within A *)
       ... t.x   ..     (* is illegal since t is qualified by B *)

      end B;
    begin

     ... v.x+y.x ....                  (* is legal *)

    end A;
```

```
begin (* outside A *)

  ... v.z ..          (* is illegal since hidden, and so close as well *)
  ... v.y.x ...       (* is illegal since x is close *)
end
```

6.3.

6.2 Static location

We say that the occurrence of an identifier W is in a unit M if M is the syntactic unit most tightly enclosing that occurrence. On the basis of the program structure every occurrence of an identifier W in a unit M can be unequivocally related to a unit N, where the corresponding attribute W is defined. The unit N is called the static container for that occurrence of W in M and is denoted by SC(W, M). More precisely, by a static container of an occurrence of an identifier W in a unit M we mean a syntactic unit N such that:

- W is defined in N
- there exists a unit P satisfying the following conditons:

(1) N belongs to the prefix sequence of P (or is P), (2) M is declared in P directly or indirectly, (3) there is no other unit closer to M than P satisfying (2) in which W is an attribute, (4) N is P's nearest prefix defining W (5) if W is illegal (hidden or not taken) in P, then the static container is undefined.

The following figure illustrates this definition

the prefix sequence of P

```
P <-------- R  <------------  SC(W,M)=N ... declaration of W ...
\^
|
.
.
.
\^
|
M ...   the occurrence of W ...
```

The static location of an identifier W is defined for the occurrence of W in a unit M iff there exists a static container SC(W, M). Every program must be an expression in which the static location is defined for all occurring identifiers. The static container is sufficient to determine the static attribute of a unit (constant).

Example.

Consider the following program

```
block
  unit M: class; var X ... end M;
  unit N: M class; var X ... end N;
  begin
    pref N  block (* P *)
    var Y : ...;
```

```
unit R: class;
   ... X ... Y ...
end R;
begin
  new R;
  ...
  pref N  block (* S *)
  var Y : ...,
  unit T: R class;
    ... X ... Y ...
  end T;
  begin
    new T;
    ...
  end S;
end P;
end
```

Here we have

```
  SC(X, R)=SC(X, T)=N
and SC(Y, R)=P, SC(Y, T)=S.
```

6.3 Objects

An object O of type M with the prefix sequence M1, ..., Mk=M (k=>1) is:

- a k-tuple of the form O = (<V1, M1>, ... <Vk, Mk>) where Vi is the valuation of non-static attributes defined in the unit Mi, - and a unique reference pointing to this k-tuple.

Since the references are unique, two objects are different even if their tuples are identical.

Now let us define the valuation of an attribute of object O, depending on the kind of that attribute:

- the valuation of variables and variable parameters gives their values, - the valuation of an attribute which is a subprogram is the text of its declaration and an environment. (The environment is the object containing the declaration of the subprogram. In the case of a formal subprogram the value is determined by the actual one (see 9.1.2.). The case of virtuals is discussed below.) - an attribute which is a type has the value of the form: (arrayof)<j> text of declaration.

6.3.1 Virtual attributes

The main feature of virtual atributes is that a redeclaration of an identifier denoting a virtual subprogram in a prefixed unit does not cover the declaration in the prefix but replaces it in all occurrences. The replacement takes place in the so-called virtual chains of identifiers. We define this notion below. Let F be a subprogram and M a unit. By a virtual chain of F in M we mean a sequence of virtuals corresponding to the maximal subsequence N1, ..., Nk of the prefix sequence of M such that:

(1) F is a legal identifier in every Ni and denotes an attribute specified as virtual (unit virtual F: ...) (2) In all the units Ni except Nk, F must not occur in the hidden list (3) In all the units except N1, F must occur in the taken list unless the list is not specified. F must not occur in the taken list in N1 if the list is specified. (4) After removing the declaration of F from N1, F should be an illegal attribute in N1 (hidden in the prefix, not taken) or should denote a non-virtual attribute (5) If Nk is not M, then one of the following conditions must be satisfied: - F occurs in the hidden list in Nk, - F does not occur in the taken list in the unit prefixed directly by Nk if the list is specified, - F is redefined in the unit prefixed directly by Nk as a non-virtual attribute (then it must not occur in the taken list either). The class Nk from the definition is called the end of the virtual chain. For a given unit and an identifier there may exist more than one virtual chain.

Example 6.1.

```
M      unit  virtual F: <M-body>

N      unit  virtual F: <N-body>

P             ....  F  ....

R      unit  F: <R-body>

S      unit  virtual F: <S-body>
       hidden F;

T      unit  F: <T-body>
```

We have three virtual chains of F with respect to T. One is for F from the classes M and N:
(F: <M-body>), (F: <N-body>),
The second is for F from the class S:
(F: <S-body>)
And the third one is for F in T:
(F: <T-body>)

Restrictions

(i) All virtual attributes belonging to the same virtual chain must be of the same kind (either function or procedure),

(i) All the declarations of the virtuals belonging to the same virtual chain must have formal parameter lists of the same pattern, in particular:

- the lists may use different names of formal parameters, but the correspondence between formal types must remain valid,
- the class types of corresponding formal variables or functions must belong to the same prefix sequence,
- the types of variable parameters or formal functions in the ending of the virtual chain must not be less strongly defined than the types of the corresponding parameters in the beginning, i.e., a formal or system type against a statically defined type,

- the types of virtual functions must be identical or the type of the function from the beginning of the virtual chain must be a prefix of the type of the function from the ending,

(iii) The compatibility of virtuals must be defined statically.

Example 6.2. *(1) The following lists are not compatible*

```
.... (type T; type P;  X: T; Y: P) ....
 .... (type R; type S; X: S; Y: R) ....
```

(2) The following lists are compatible iff M and N belong to the same prefix sequence (and both are classes)

```
.... (type T; Z: T; Z1: M) ....
 .... (type P; X: P; Y: N) ....
```

(3) The following lists are compatible iff M denotes a system type (coroutine or process) or is a formal type

```
at the beginning:  (X: M; Y: real)
at the ending:     (X:coroutine; Y:real)
```

(4) The following lists are not compatible:

```
... (Y:integer)
... (Y:real)
```

(5) The lists of the function from the beginning of the chain

```
... function (Z:integer; Z1:P) : M
```

and from the ending

```
... function (Z:integer; Z1:P) :N
```

are compatible iff M is a prefix of N.

6.4.2.

6.3.2 Valuation of virtuals

Let O be an object of type M with the prefix sequence M1, ..., Mk=M. The value of virtual attribute F declared in Mi is:

- the text of the declaration taken from the end of the virtual chain,

- the environment of the object O.

Example 6.3. *An object of the class T given in the example 1 from 6.4.1 is of the following form:*

The name "virtual subprogram" is derived from the features of virtual entities, i.e., in any class a virtual subprogram F with an empty statement list can be declared and then used as a virtual entity within the body of the class. The user can assume the results of its execution without knowledge of its internal structure. He can declare in a subclass a virtual subprogram F again. This declaration replaces the previous one. So, during the calls of the subprogram F in the body of the class as well as in the body of the subclass, the subprogram with the text defined in the subclass will be executed. This replacement is done only if F is a virtual attribute of the subclass. Otherwise the new declaration of F covers the virtual attribute of the class.

Abstention from those rules permits us:

(i) to define the types of the parameters of a virtual subprogram and to check them already at compilation time,

(ii) to execute the virtual subprogram declared at the beginning of the prefix sequence; its body may be empty, but it is always defined,

(iii) to end the virtual chain and to cover a virtual identifier by a redeclaration.

The possibilities (ii) and (iii) can be used in the following case. Let M and N be system classes of the form :

```
unit M: class;
  unit virtual error: procedure;
   (* virtual procedure to be defined in M's subclasses*)
  end error;
```

```
begin
   ...
   if B1 then call error fi;
end M;

unit N:  M class;
  unit virtual error: procedure;
          (* the definition of the body of error. It
           will be executed during the calls within N
           as well as in M *)
  end error;
begin
   ...
   if B2 then call error fi;
end N
```

If the programmer prefixes his own units by class M, he can declare his own virtual procedure error. If he does not intend to signalize any errors, he is able to use M without a redeclaration. Then if the condition B1 is satisfied, the procedure with an empty body will be called, i.e., no statement will be executed. On the other hand, if the programmer uses N as a prefix of his own units, he can redeclare his own non-virtual procedure error. In consequence, during the execution of statements of the classes M and N the procedure defined by this system in the class N will be executed. However during the execution of the user's units the procedures defined by himself will be executed.

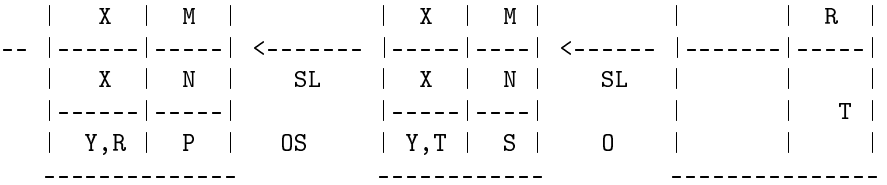
6.4 Dynamic location

An executable program must always be a well-formed expression. During its execution we can deal with many objects of the same syntactic unit even at the same time. Hence an execution of a statement (in an object) requires identification and access to all the syntactic entities used. In order to define the syntactic environment of object O (of unit M) a static link (SL) is introduced. This link always points to an object O1 of a unit N such that M is declared in N. Let us consider the occurrence of an identifier W within a body of class N from the prefix sequence of M. Let SL(M) denote the SL-chain of objects starting from an object of unit M. This means that SL(M) is a sequence of objects O1, ..., Ok such that O1 is an object of unit M, Ok is an object of the main program, the SL-link of object Oi points to object Oi+1, for every i=1, ..., k-1.

The dynamic container of the occurrence of W in a body of class N with respect to an object O1 (denoted by DC(W, N, O1)) is an object Oi from SL(M) such that:

(*) Oi is an object of the unit prefixed by the static container SC(W, N);
(**) Oi is the nearest object in the SL-chain such that Oi satisfies (*).

Hence the dynamic container is the unique object which contains the valuation of the entity W related to the occurrence of this entity. Let us return to the example from 6.3.; after the creation (new T) of an object O of the class T the SL-chain of O is as follows:



Because SC(X, R)=SC(X, T)=N , we have DC(X, R, O)=DC(X, T, O)=OS. Since SC(Y, T)=S , we have DC(Y, T, O)=OS. On the other hand SC(Y, R)=P and DC(Y, R, O)=OP. The syntactic environment of an object is determined by the SL chain. Its main property is that for each identifier occurrence in the statements of the given object exists its dynamic container in the chain. In order to define the dynamic location of identifier W occurring in object O of unit M in a body of unit R (which belongs to the prefix sequence of M), the following steps are performed:

- a static container N=SC(W, R) is defined; - a dynamic container O1=DC(W, R, O) is defined (in the SL chain of object O, the nearest object O1 is found such that this object has a "layer" <V, N>); - a valuation V1(W) is found in the layers <V1, N1> of the object O1 such that N1 is the nearest N's prefix.

Chapter 7

Consistency of types

In order to determine the context-sensitive correctness of an assignment statement and parameter transmission it is necessary to introduce the notion of the static consistency of types. Nevertheless this notion is not sufficient to determine the correctness of the executions of those constructs. Therefore, the notion of the dynamic consistency of types will be introduced to define the semantic correctness of program. The introduced distinction follows from the implementation constraints; namely, static consistency is verified at compilation time, dynamic consistency is verified at run time.

Static consistency of types

The type (arrayof)^{i}T is statically consistent with the type (arrayof)^{j}S, where T and S are not array types, iff one of the following conditions holds:

- i=j and T=S,
- i=j=0 and T, S are integer or real types,
- both T and S are formal types,
- T is a formal type, S is not a formal type and i<=j,
- S is a formal type, T is not a formal type and j<=i,
- i=j=0 and T, S are generalized class types and T pref* S or S pref* T,
- i=j=0 and T and S are one of them a system type and the other a generalized class or system type.

Dynamic consistency of types.

The type (arrayof)<i>T is dynamically consistent with the type (arrayof)<j>S, where T and S are not array types, iff one of the following conditions holds:

- - i=j and T=S,
- - i=j=0 and T, S are integer or real types,

- - i=j=0 and T, S are generalized class types and T pref* S,
- - i=j=0, T = coroutine, and S is declared as:
  unit S: ... coroutine ...; or
  unit S: ... process .....; or
  unit S: R class..., where T is dynamically consistent with R,
- - i=j=0, T = process, and S is declared as:
  unit S: ... process .....; or
  unit S: R class..., where T is dynamically consistent with R.

At run time all formal types are replaced by actual non-formal ones. Therefore, there is no reason to define dynamic consistency for formal types. Dynamic consistency is a subrelation of static consistency. Thus the dynamic consistency is checked at compilation time, if possible. In other cases the check is made at run-time. From now on we shall use the following notation:
- for the desription of context properties, an occurrence of an expression E is considered to be contained in the body of unit M,
- for the desription of semantic properties, an occurrence of an expression E is considered to be contained in the body of unit M, with respect to an object O of type M0 such that M pref* M0.

8.

Chapter 8

Expressions

Expressions are composed of primitive expressions - constants and variables by means of system operators and functions. They serve as patterns for computing a certain value. Two kinds of expression properties have to be considered: context (static) and semantic (dynamic) ones.

CONTEXT PROPERTIES. We consider two context properties of each expression:

- to be a well-formed formula,
- to have a static type.

The context correctness of an expression is examined at compilation time. From now on, an expression is said to be a well-formed formula (shortly : WFF) if it is statically correct. The static type of an expression is determined by the program text.

SEMANTIC PROPERTIES. We consider three semantic properties of each expression:

- to be defined, i.e. to have a value,
- to have a dynamic type,
- to have the type of its value.

In some cases (for expressions of formal types) type must be determined at run-time. Replacing formal types by the corresponding actual ones in the static types of expressions, we obtain the dynamic types of those expressions. Notice, that the actual type may not be accessible, if the dynamic container for the formal type of the expression was killed. The dynamic type will be defined only for the expressions which may occur on the left side of an assignment, i.e., for variables. When the value and the type of the value are computed, the semantic correctness of the expression is established. From now on an expression is said to be defined if it is dynamically correct at run-time. The correctness of an expression will be examined under the assumption that it is a WFF. Five kinds of expressions are distinguished: arithmetic, boolean, character, string, and object expressions.

8.1 Constant

SYNTAX.

```
          <constant>:

      -----> <identifier> ----->
```

CONTEXT.

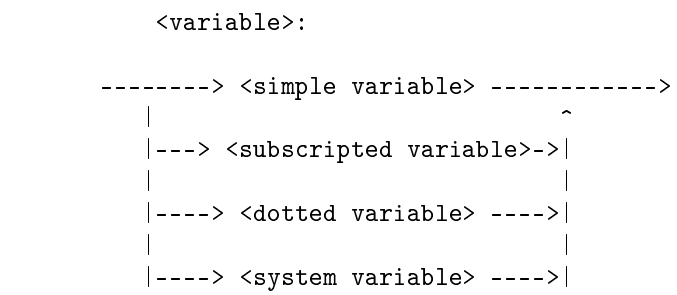
Let E be a constant Q. The expression Q is a WFF if the static container SC(Q, M) exists. The static type of Q is determined by its declaration (see 5.1.). A constant cannot occur on the left side of an assignment statement, as an actual output parameter, or in an expression X.Q, where X is an object expression.

SEMANTICS.

The constant Q is always defined. The value of the constant is fixed from the declaration of that constant and cannot be modified. The type of the value is equal to the static type.

8.2 Variable

SYNTAX.



For each kind of variables its context and semantic correctness will be defined. Additionally the dynamic address of a variable will be defined as a pair: (reference to an object, attribute of that object).

8.2.1 Simple variable

SYNTAX.

```
          (simple variable>:

      ----> <identifier> ----->
```

Let E be a variable Z.

CONTEXT. The variable Z is a WFF if the static container SC(Z, M) = R exists. The static type of Z is determined by the declaration of Z and may be a formal one.

SEMANTICS.

The variable Z is defined if the dynamic container O1 = DC(Z, M, O) exists. Let the static type of Z be: (arrayof)<i>S. The dynamic type of Z is equal to (arrayof)<i>S in the case where S is not formal, otherwise it is (arrayof)<i+k>T, where the actual type corresponding to the formal one is (arrayof)<k>T. The actual type is taken from the dynamic container DC(S, R, O1), i.e., from an object belonging to the SL chain of the object O1. The value of Z is given by the corresponding valuation of Z in the object O1. The address of Z is a pair: (the reference to O1, attribute Z of O1).

8.2.2 Subscripted variable

SYNTAX.

```
          <subscripted variable>:

 --> <simple variable> --> ( -> <arithmetic expression> -----> ) -->
                           ^                                  |
                           |<----------- , --------------|
```

Let E be an expression of the form Z(A1, ..., Ak), where Z is a simple variable and A1, ..., Ak are arithmetic expressions.

CONTEXT.

Let (arrayof)<i>S denote a static type of Z. The expression Z(A1, ..., Ak) is a WFF if: - Z and A1, ..., Ak are WFFs, - static types of A1, ..., Ak are integer or real, - 1<=k<=i. The static type of E is (arrayof)<i-k>S.

SEMANTICS.

The expression E is defined if: - the expression Z(A1, ..., Ak-1) is defined and its value equals the reference to a non-empty array object O1 with the bounds l and u, l<=u. - the value of Ak is defined and its truncation l1 satisfies: l<=l1<=u.

The dynamic type of E is equal to the static one if S is not formal, otherwise it is (arrayof)<i-k+j>T where the actual type corresponding to the formal one is (arrayof)<j>T. The actual type is determined as for a simple variable (see 8.2.1.). The value of E is that of the attribute (l1) of the object O1. The address of E is the pair: (the reference to O1, attribute (l1)).

8.2.3 Dotted variable

SYNTAX.

```
          <dotted variable>:

      -> <qualified object expression> -->. --> <variable> ---->
```

It is sufficient to consider the expression E of the form X.Y, where Y is a simple or subscripted variable.

CONTEXT.

The expression E is a WFF if:

- X, Y are WFFs, X is the qualified object expression, - the static type of X is a generalized class type, - Y is a non-closed attribute of the static type of X.

The static type of E is the same as the static type of Y. Notice that the static type of X cannot be a formal type.

SEMANTICS.

The expression E is defined if:

- the expression X is defined, - the value of X is a reference to a non-empty object O1.

The dynamic type of E is the same as the dynamic type of Y would be if Y occurred in the object O1. The value of X.Y is that of the attribute Y of the object O1. The address of X.Y is the address of Y would be if Y occurred in O1.

8.2.4 System variable: result

textscSyntax.

```
    <system variable>:

------> result ---------------------------------------->
```

CONTEXT and SEMANTICS.

For every function F there is an implicitly declared variable result of type T of the value of function F. The initial value of that variable depends on type T (is equal to the default value of type T), the final value (after completion of a function call) is also the value of function F for the given call (see 9.1.2.). An occurrence of the variable result is matched with the smallest unit F which contains that occurrence and which is a function.

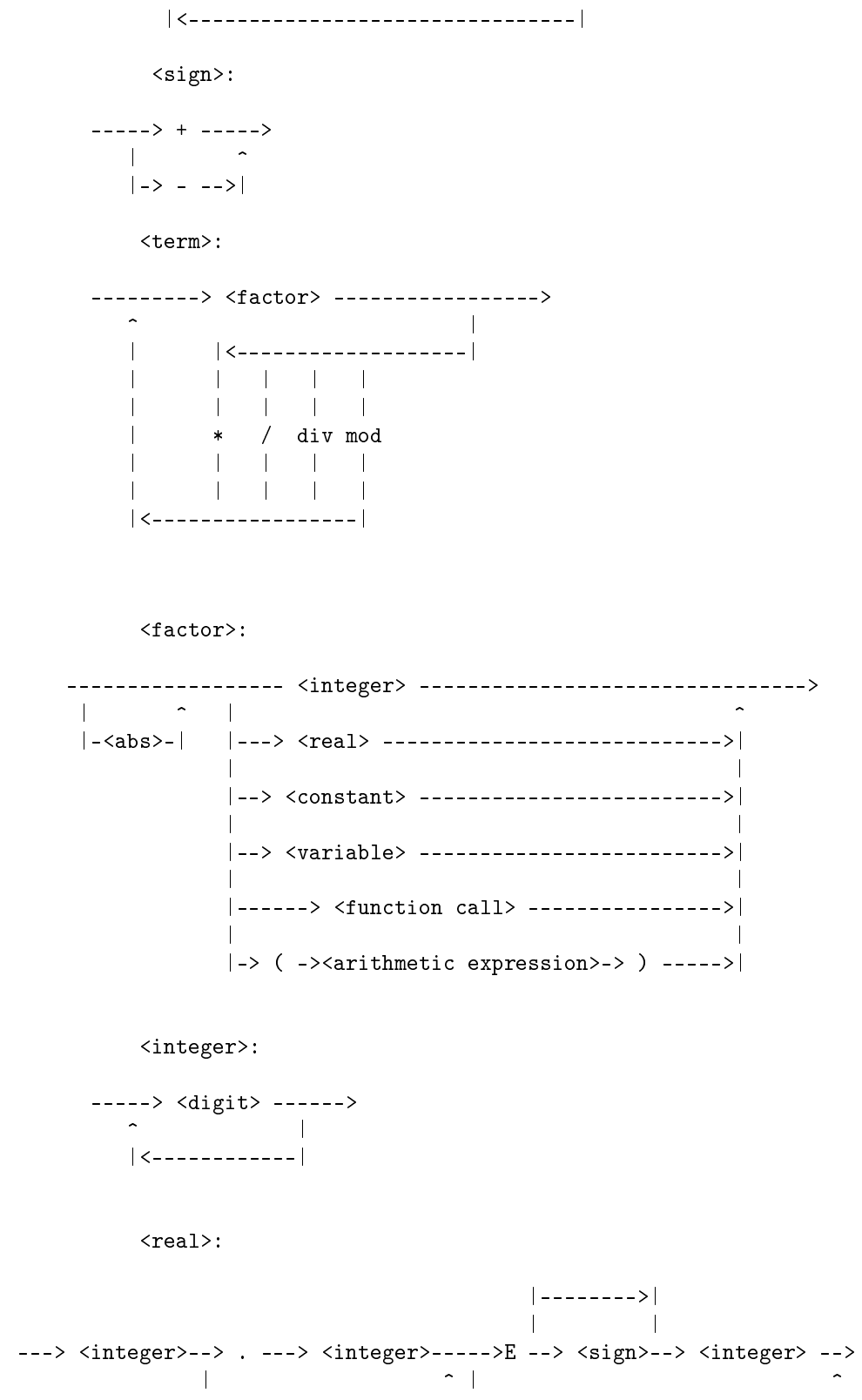
Example.

```
unit Newton_symbol: function (i, k:integer): integer;
var j: integer;
begin
   if  i>= k and k>=0
   then result:=1;
     for j:=0 to k-1
     do
       result:=result*(i-j)div(j+1)
     od
   fi
 end Newton_symbol;
```

8.3 Arithmetic expression

SYNTAX.

```
    <arithmetic expressionXE "arithmetic expression"§>:

   |------------------->|
   |                    |
-----------> <sign> --------> <term> ------->
      ^                                  |
```


|<-------------------------------|
<sign>:
-----> + ----->
| ^
|-> - -->|
<term>:
---------> <factor> ----------------->
^ |
	<-------------------
* / div mod
<-----------------
<factor>:
----------------- <integer> ------------------------------->
| ^ | ^
|-<abs>-| |---> <real> ---------------------------->|
| |
|--> <constant> ------------------------->|
| |
|--> <variable> ------------------------->|
| |
|------> <function call> ---------------->|
| |
|-> (-><arithmetic expression>->) ----->|
<integer>:
-----> <digit> ------>
^ |
|<------------|
<real>:
|-------->|
| |
---> <integer>--> . ---> <integer>----->E --> <sign>--> <integer> -->
| ^ | ^

```
                    |------------------>|  |--------------------------->|
```

(function call will be defined in 9.1.2.).

CONTEXT and SEMANTICS.

The type of the value of an arithmetic expressionXE "arithmetic expression"§ is always equal to its static type. The dynamic type is not to be defined. The context and semantic properties of arithmetic expressions will be defined inductively.

Factors.

The type of an integer is integer, the type of a real is real, their values are given directly. Constant, variable, and function call must be WFFs (in the meaning of 8.1., 8.2 and 9.1.2.), and of type integer or real (in order to create a well-formed factor). The factor is defined iff the variable and the function call are defined. The context and semantic properties of the factors of the form " abs A1 ", and " (A2) " are the same as those of arithmetic expressions A1 and A2, respectively. The value of " abs A1 " is the absolute value of A1.

Terms.

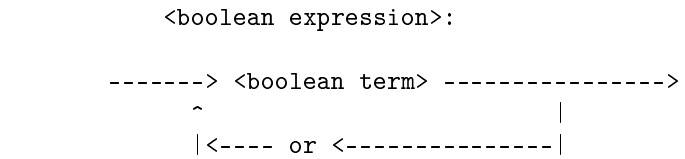
The operators *, /, div, mod are interpreted as multiplication, division, integer division and remaindering, respectively. The last two operators are defined for integer arguments only, " A1 div A2 " is equal to the truncation of A1/A2; " A1 mod A2 " is equal to the remainder of A1/A2. The type of a term of the form <factor> <operator> <factor> is real if the operator is /, or at least one of the arguments is of type real. The term " A1/A2 " is defined if the value of A2 is different from 0. The value is defined inductively if Av1 and Av2 are the values of factor A1 and term A2 respectively, and <W> is an interpretation of operator W, then the value of a term of the form " A1 W A2 " is Av1 <W> Av2. If one of the arguments is of type integer and the other is of type real then for the operators *, / the integer type value is converted into a real type one.

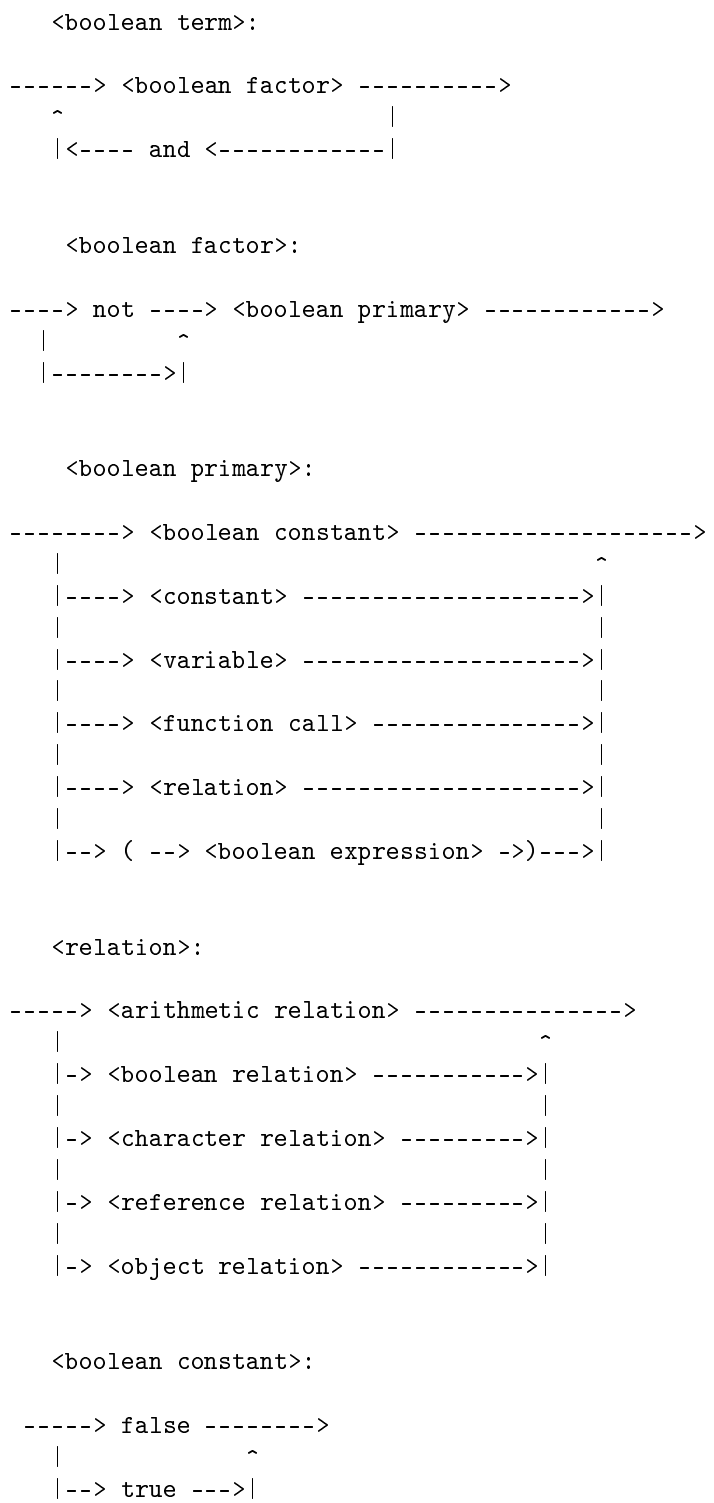
Arithmetic expression.

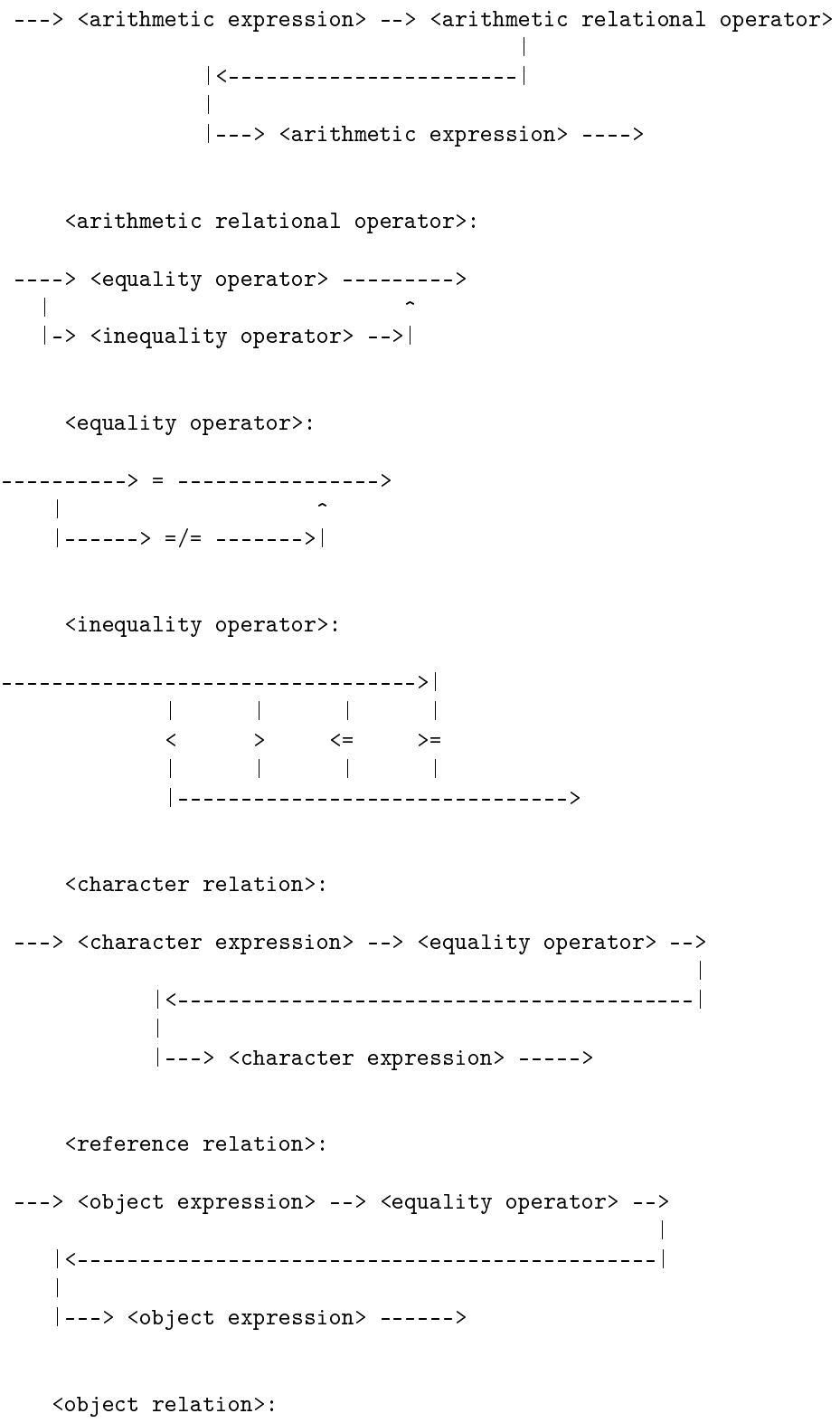
An arithmetic expression of the form <term> <sign> <term> is of type integer if both terms are of that type and it is of type real in the opposite case. A value is defined inductively: if Av1 and Av2 are the values of term A1 and arithmetic expression A2, respectively, then the value of an expression A1+(-)A2 is Av1+(-)Av2, the value of +(-)A1 is +(-)Av1. If one of the arguments is of type integer and the other is of type real, then the integer type value is converted into a real type one.

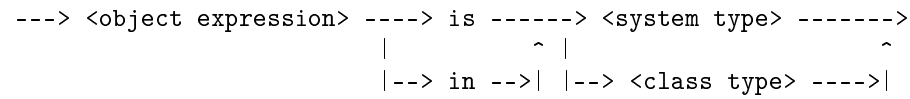
8.4 Boolean expression

SYNTAX.




<boolean term>:
<boolean factor> ----------
and
<boolean factor>:
not
<boolean primary> ------------
<boolean primary>:
<boolean constant> --------------------
<constant> --------------------
<variable> --------------------
<function call> ---------------
<relation> --------------------
(--> <boolean expression> ->)
<relation>:
<arithmetic relation> ---------------
<boolean relation> -----------
<character relation> ---------
<reference relation> ---------
<object relation> ------------
<boolean constant>:
false --------
true
<arithmetic relation>:


---> <arithmetic expression> --> <arithmetic relational operator>
|<------------------------|
|---> <arithmetic expression> ---->
<arithmetic relational operator>:
----> <equality operator> --------->
|-> <inequality operator> -->|
<equality operator>:
----------> = ---------------->
|------> =/= ------->|
<inequality operator>:
---------->|
< > <= >=
|-------------------------------->
<character relation>:
---> <character expression> --> <equality operator> -->
|<--|
|---> <character expression> ----->
<reference relation>:
---> <object expression> --> <equality operator> -->
|<---|
|---> <object expression> ------>
<object relation>:



CONTEXT and SEMANTICS.

The context and semantic properties of boolean expressions can be defined in the same way as those of arithmetic ones. A boolean expression is of type boolean.

Boolean primary.
The value of a boolean constant true and false is T and F, respectively. The equality and inequality operators have the usual interpretation. Let A1, A2 be two defined arithmetic expressions and let Av1, Av2 be their values. Let <W> be an interpretation of the arithmetic relational operator W. Then the value of arithmetic relation " A1 W A2 " is Av1 <W> Av2. If one of the expressions is of type integer and the other is of type real then the integer type value is converted into real type one.

Let C1, C2 be two defined character expressions and let Cv1, Cv2 be their values. Then the value of the character relation " C1=C2 " (" C1=/=C2 ") is true iff the characters Cv1, Cv2 are identical (different). For string type there are no relations, even no equality.

A reference relation " X1=X2 " (" X1=/=X2 ") is a WFF if X1 and X2 are well-formed object expression. The static types of the expressions have to be statically consistent. The relation is defined if X1 and X2 are defined. The value of that relation is true iff the values of both expressions are equal to (different from) the same reference; in particular, if they are both equal to none, then the value of " X1=X2 " is T. An object relation "X is S" is a WFF if S is a generalized class identifier, X is a WFF, and the static type of X is statically consistent with S. An object relation "X in S" is a WFF if S is a generalized class or system type identifier, X is a WFF, and the static type of X is statically consistent with S. The value of the relation "X is S" is T iff the value of the expression X is the reference to an object of class S. The value of the relation "X in S" is T iff the value of X belongs to the set |S| .

Boolean factor.
The value of a boolean factor "not B", where B is a boolean primary, is T iff the value of B is F.

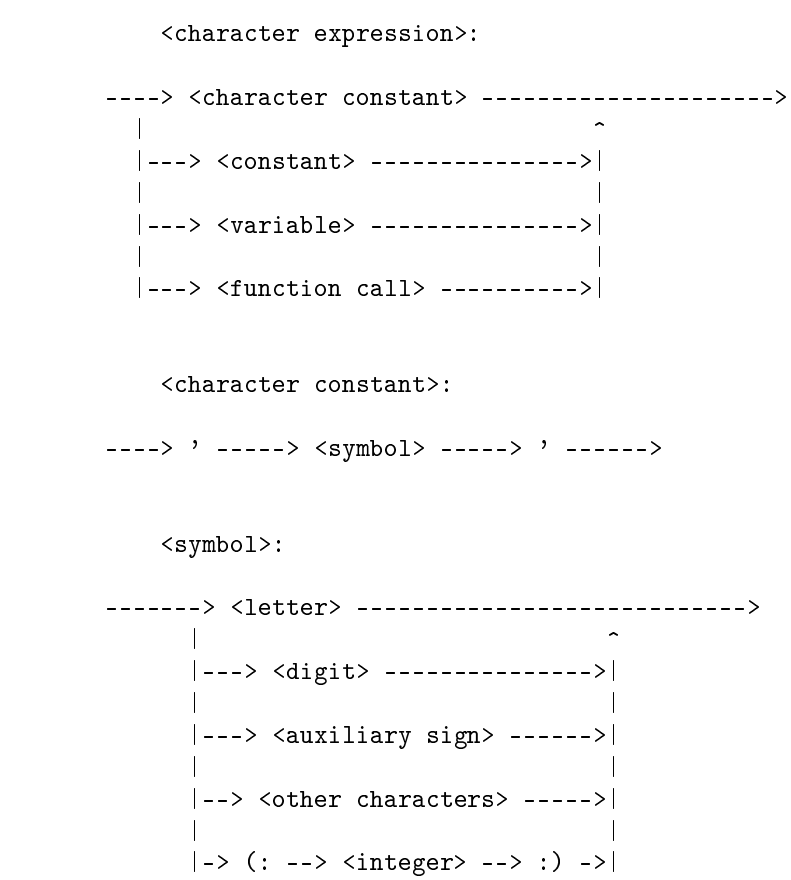
Boolean term.
Let Bv2 and Bv1 be the values of boolean factor B2 and boolean term B1, respectively. Then the value of a term of the form "B1 and B2" is T iff Bv2=Bv1=T.

Boolean expression
Let Bv1 and Bv2 be the values of boolean term B1 and boolean expression B2, respectively. Then the value of an expression of the form "B1 or B2" is F iff Bv1=Bv2=F.

The value of the arithmetic and boolean expression is computed from left to right with the following operator priorities: (1) parentheses (,), abs (2) *, /, div, mod (3) +, - (4) <, <=, >, >=, =, =/= (5) not (6) and (7) or.

8.5 Character expression

textscSyntax.

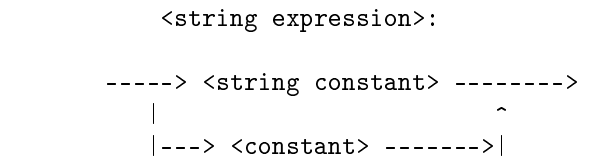


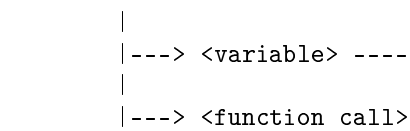
Context and Semantics.

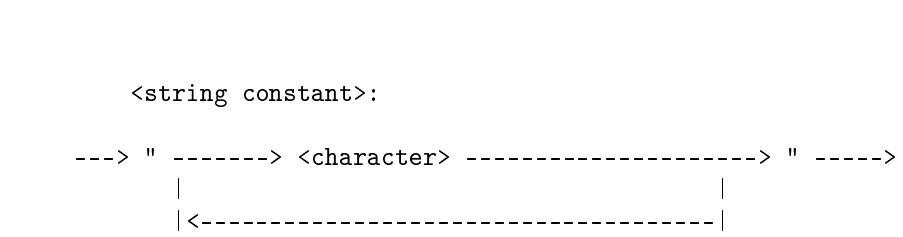
Constant, variable and function call are WFFs if they are of type character. The standard function ord is defined for a character expression and gives an integer value (dependent on implementation).

8.6 String expression

Syntax.







SEMANTICS.

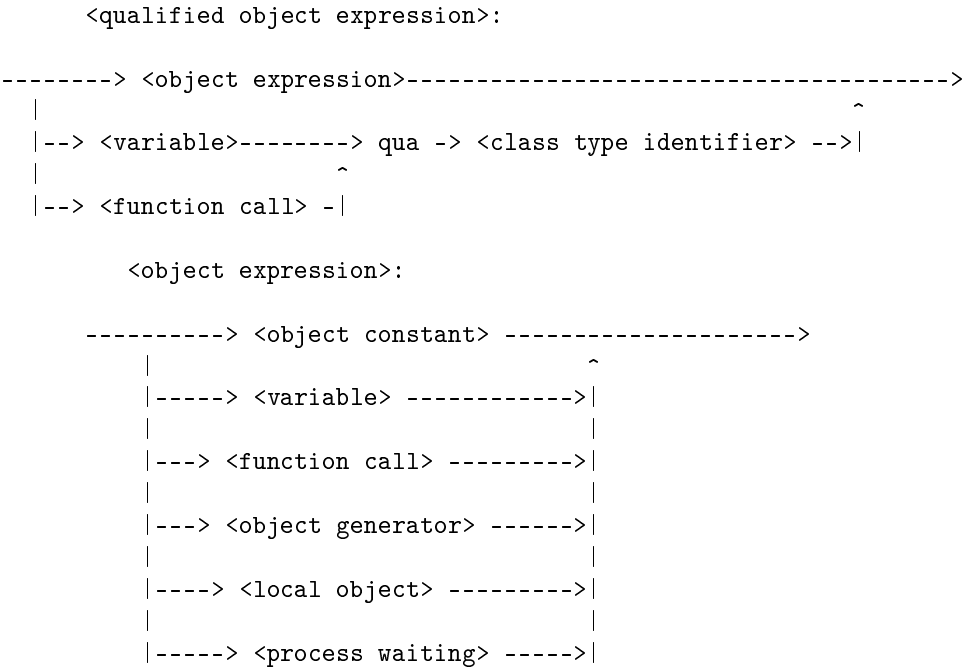
Constant, variable and function call are WFFs if they are of string type. The quotation mark " in the string constant is written twice "".

Remark The string type is a constant type in the sense that the universe is defined at compilation time and there are no string operations predefined in the language. However, a standard function may transform a string into an array of characters. Then the user can treat the array of character as a text type and can define any set of suitable text operations.

End of remark

8.7 Object expression

SYNTAX.



```
    <object constant>:

-----> none -------- >

    <local object>:

----> this ----> <class type> ---------->
```

(Function call and object generator will be defined in 9.1.2, process waiting will be defined in 11.1. Variable is described in 8.2.).

CONTEXT. The constant none is of a fictitious type statically consistent with any non-primitive type. To define the context of a local expression let us recall that the occurrence of the expression E is considered in the unit M. Let E be the local object "this T", then E is a WFF if there exists a unit N such that M decl* N and T pref* N, (i.e., there exists a unit N statically enclosing M and containing T in its prefix sequence). The static type of the expression E is T. The qualified object expression of the form "X qua T" is a WFF if X is a WFF and the static type of X is statically consistent with T. The static type of this expression is T.

SEMANTICS. The constant none is always defined as an empty object. Every compound and system type is dynamically consistent with the fictitious type of none. The value of the local object "this T" is the nearest object of the type T1 belonging to the SL chain of the object O such that T1 is prefixed by T, (recall that O contains the given occurrence of the local object). The expression "this T" is defined if its value exists. The dynamic type is not to be defined. The type of the value is S. The qualified object expression of the form "X qua T" is defined if X is defined, its value is different from none, and the dynamic type of X as well as the type of its value are dynamically consistent with T. The value of this expression is equal to the value of X. The dynamic type is not to be defined.

Chapter 9

Sequential statements.

Sequential statements are patterns for the sequencing of primitive actions.

SYNTAX.

```
        <sequential statement>:

    --------> <primitive statement> ------------>
       |                                  ^
       |-------> <compound statement> ---->|
```

In a similar way to that followed in the description of expressions we shall consider context and semantic properties of statements. A statement will be called a WFF if it is correct at compilation time, and said to be defined if it is correct at run time.

9.1 Sequential primitive statements

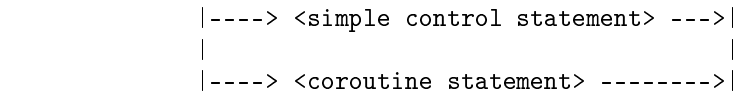
The result of an execution of a primitive statement consists in the modification of:

- the valuation (assignment statement);
- the configuration (allocation and deallocation statementXE "allocation statement"§);
- the control (control statement).

By a configuration we mean the set of all objects existing at a given state of computation.

SYNTAX.

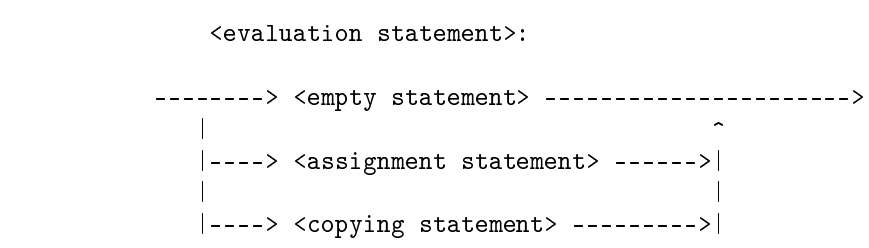
```
        <primitive statement>:

    --------> <evaluation statement> ------------->
       |                                     ^
       |----> <configuration statement> ---->|
       |                                     |
```

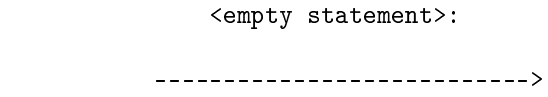


9.1.1.

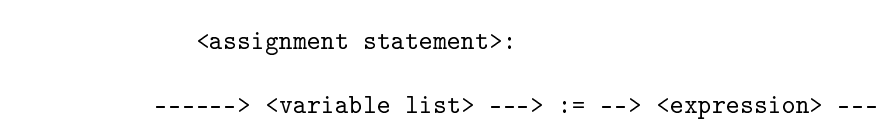
9.1.1 Evaluation statement

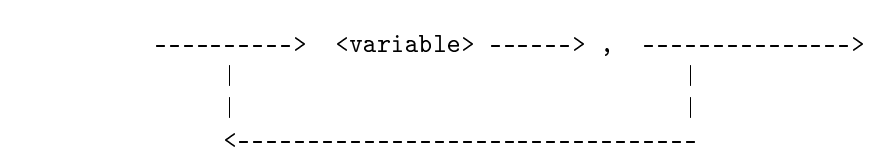
Syntax.





Semantics.
An execution of an empty statement leaves the overall state of computation not changed.
Syntax.





Context.
An assignment statement of the form y1, ..., yk:=e is a WFF if:

- variables y1, ..., yk and expression E are WFFs;
- the static types T1, ..., Tk of variables y1, ..., yk are statically consistent with the static type S of the expression E.

SEMANTICS.

The execution of the statement consists of three steps : prologue, body and epilogue.

In the prologue the computation of the addresses of variables y1, ..., yk is performed, i.e.:

- For a dotted variable of the form X.z, the value of the expression X is computed;

- For a subscripted variable of the form Z(i1, ..., ij) the value of the expression Z(i1, ..., ij-1) is computed. If Z is of a formal type, then the dynamic type T of the variable Z is determined. Finally the value of the expression ij is computed.

The above actions are performed from left to right.

During the body the computation of the type and the value of expression E is performed.

The epilogue checks if the statement is well-defined and assigns the values to the attributes determined by the addresses evaluated during the prologue.

An assignment is defined, if: - the expressions y1, .., yk, E are defined; - the dynamic types of y1, .., yk are defined and are dynamically consistent with the type of the value of E.

The values are assigned from right to left, i.e., at first the value of E is assigned to yk (with possible conversion to the type of yk), next the value of yk is assigned to yk-1 (with appropriate conversion), and so on.

For example, when r is real, n is integer, then:

after r, n:=2.5 we have n=2, r=2.0, after n, r:=2.5 we have r=2.5, n=2.

Remark.

The value of the expression Z computed at prologue may point to a non-empty object O, but it could be changed to none as a result of the deallocation of the object O (during the execution of the statement). This will be detected at epilogue and will result in a run-time error.

End of remark.

An object of a compound type can be simultanously referenced by a number of variables. If X and Y are the variables of such a type, then after assignment X:=Y, both variables reference the same object. Hence some side-effects may occur: the value of an attribute of the object referenced by variable X can be changed as a result of an access to that object by means of variable Y. In order to avoid such effects, one can use a copying statement:

X:=copy(Y)

after which both variables reference identical objects but not the very same one.

SYNTAX.

```
        <copying statement>:

-> <variable list> -> := -> copy -> ( -> <object expression> -> ) ->
```

CONTEXT and SEMANTICS.

The semantics of the copying statement differs from that of the assignment statement in the following points:

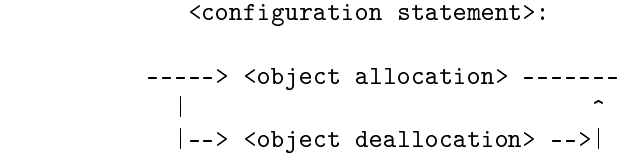
- The copying statement is defined if the value of the right side object expression E is a reference to a terminated class object (i.e., an object whose all statements were completed, see 9.1.3). Coroutine or process objects must not be copied.

- During the epilogue, the copy of the value of the expression E is assigned (a copy of none is none). 9.1.2.

9.1.2 Configuration statement

Configuration statements correspond to the generation and deallocation of units and arrays. Allocation of an array object is a result of array generation, allocation of a unit object is a result of a subprogram call, generation of a generalized class object or block statement.

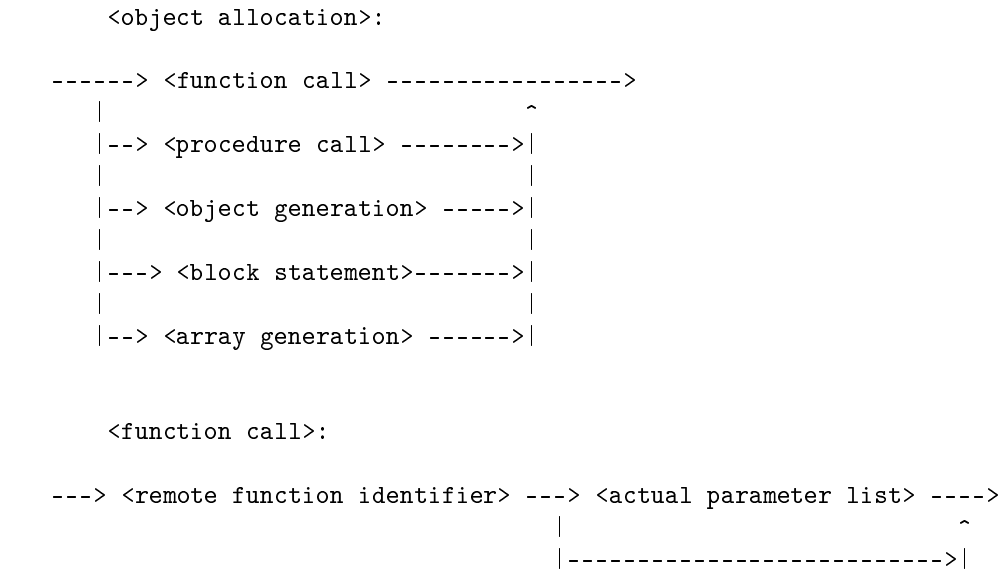
SYNTAX.

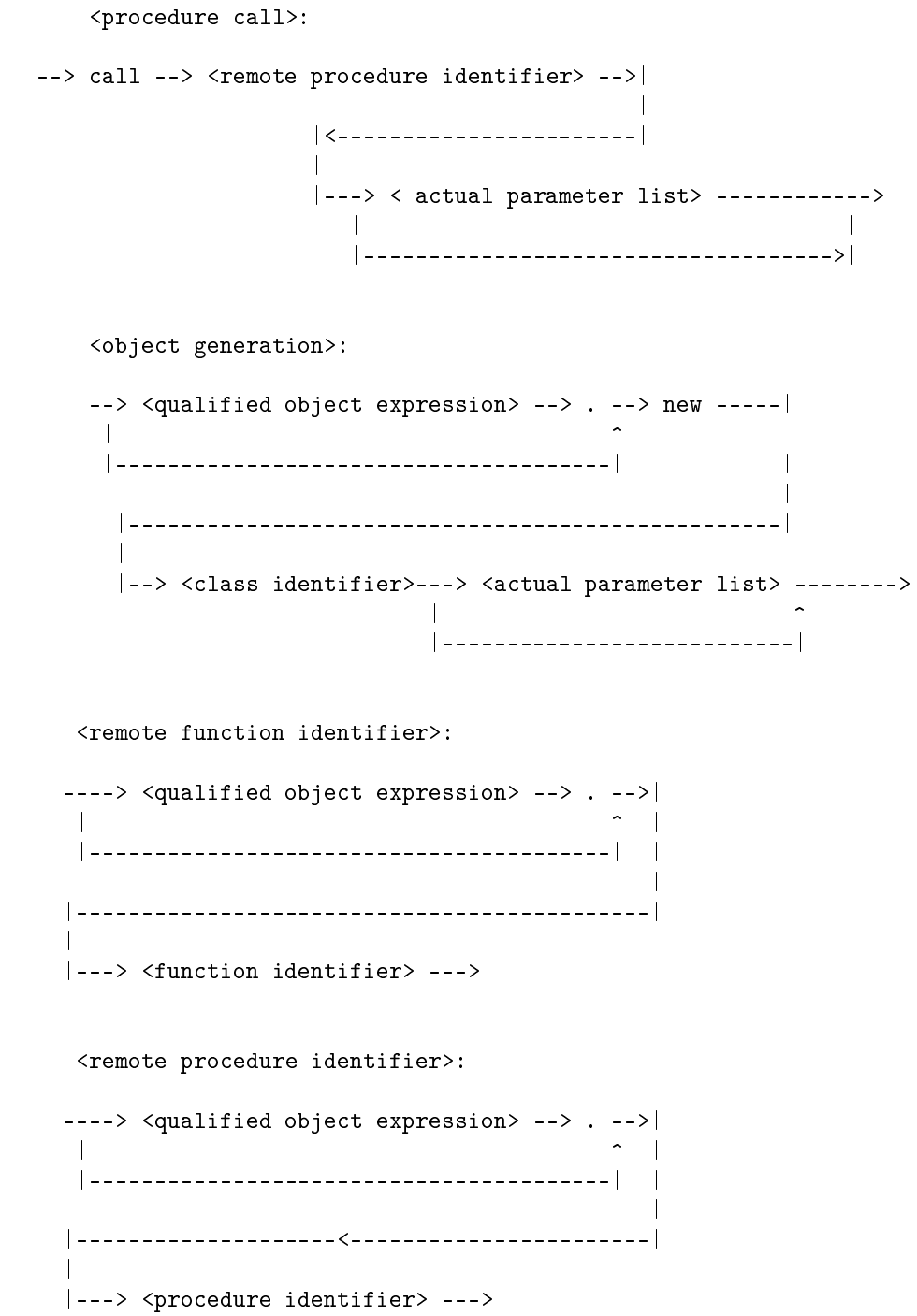


9.1.2.1.

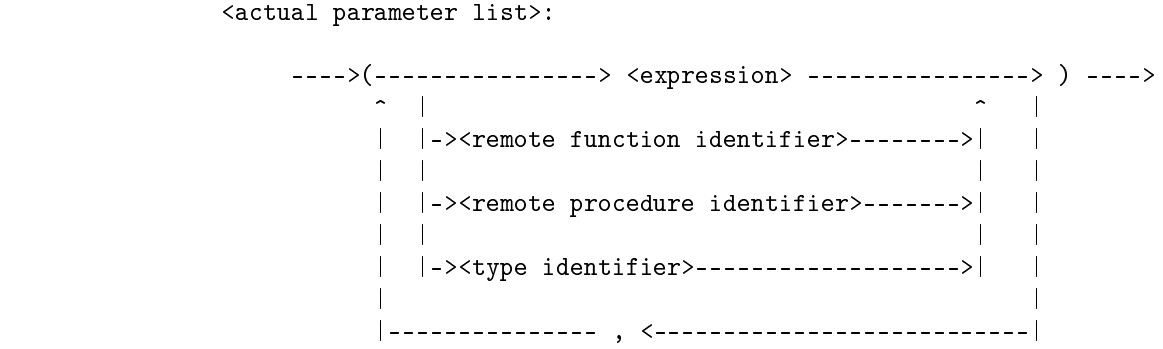
Allocation statement

SYNTAX.




<procedure call>:
--> call --> <remote procedure identifier> -->|
<object generation>:
--> <qualified object expression> --> . --> new -----|
--> <class identifier>---> <actual parameter list> -------->
<remote function identifier>:
----> <qualified object expression> --> . -->|
|---> <function identifier> --->
<remote procedure identifier>:
----> <qualified object expression> --> . -->|
|---> <procedure identifier> --->

<actual parameter list>:



CONTEXT.

We shall start with an allocation of a unit object O, i.e., subprogram call, object generation and block statement. The execution of those statements causes the generation of the new object O. Let Pa1, ..., Pak denote actual parameters, k>=0, and let X be an object expression. The allocation of an object of unit M is of one of the following forms:

- for function M: M(Pa1, ..., Pak) or X.M(Pa1, ..., Pak) (a function call must occur in an expression; it is not allowed as an independent statement);

- for procedure M: call M(Pa1, ..., Pak) or call X.M(Pa1, ..., Pak);

- for class M: new M(Pa1, ..., Pak) or X.new M(Pa1, ..., Pak); (an object generator may occur in an expression and it is also allowed as an independent statement).

- for block statement: pref M(Pa1, ..., Pak) block...end or block... end (a block can be considered as an unnamed unit and a generation of its object is the result of an occurrence of that block statement).

The allocation of a unit object is a WFF if:

- a unit identifier M is visible (in the sense of the rules used for the variables, see 8.2.), - the actual parameters are WFFs, - the formal parameter list and the actual parameter list are statically compatible in the sense given below.

Let us recall (5.3.5.) that a formal parameter list of a unit M is defined as a concatenation of the lists of units belonging to the prefix sequence of M.

Static compatibility of parameters.

The list of formal parameters (Pf1, ..., Pfj) is statically compatible with the list of actual parameters (Pa1, ..., Pak) if j=k and for i=1, ..., k the following conditions hold:

- if Pfi is an input/output formal parameter then Pai is a WFF of a static type which is statically compatible with the static type of parameter Pfi, - if Pfi is an output/inout parameter then Pai is a variable, - if Pfi is a formal function (procedure) then Pai is a function (procedure) identifier, - if Pfi is a formal type then Pai is a non-primitive type identifier.

SEMANTICS.

The allocation of a unit object O is defined if: - the unit and its environment are determined, - the list of formal parameters is dynamically compatible with

that of actual parameters (in the sense provided below), - three steps of actions, called prologue, body, and epilogue, are determined.

Note the difference between the unit identifier and the unit itself. The environment is the object which becomes the syntactic father of O. In the case of a formal subprogram, the unit identifier may be replaced by one of many existing units. Denote by O1 the object containing the given unit object allocation statement. The prologue computes the values for input formal parameters, determines the addresses of output actual parameters, and determines actual subprograms/types. The prologue is executed in the environment of the object O1. The body transfers the control to the statements from the prefix sequence of the given unit. Those statements are executed in the environment of the object O. The epilogue transmits the values of output formal parameters (in the environment of the object O1).

Unit's environment

Consider the allocation of a named unit (i.e. it is not a block). A unit identifier has one of the following forms:

(a) M, (b) X.M or X.new M .

Ad (a). Let the static location of the given occurrence of M be determined by the attribute M of the unit T. Consider three cases:

(a1) M is an attribute of T and it is neither a virtual attribute nor a formal parameter. Then the declaration of M is determined as (at compilation time) as the declaration of the attribute M of unit T. The environment of the unit M is the dynamic container of identifier M with respect to the object O1.

(a2) M is a virtual attribute of T. Then the declaration of M is determined at run-time by the dynamic location of identifier M with respect to the given occurrence (see 6.1.5.). The environment is determined as in (a1).

(a3) M is a formal subprogram of T. Then the declaration of M and its environment are taken from the dynamic container of the identifier M. The dynamic container is determined with respect to the object O1.

Ad (b). Let X be a well-formed object expression of type R, let M be a not close attribute of R, and let the expression X be defined. Denote by O2 the non-empty object of unit R0 (R pref* R0) which is pointed to by X. The cases (a1)-(a3) have to be considered in the same way as the above ones. The descriptions differ in that the environments are determined with respect to the object O2. Note that the environment of the object becomes the syntactic father of the object O.

Dynamic compatibility of parameters.

First let us note the difference between the determination of dynamic type for the actual parameter Pa and the formal parameter Pf. The dynamic type of Pa is determined in the environment of the object O1 (containing the given allocation). It means that for the formal type S the actual type is taken from the dynamic container with respect to O1. Recall that it corresponds to the determination of the valuation of identifier S in the SL-chain of O1 (according to the visibility rules) and taking the text of declaration assigned to S (cf. 6.1.5.). The dynamic type of Pf is determined in the corresponding environment. It means that for the formal type S the actual type is taken from the corresponding dynamic container. In other words, the valuation of identifier S is searched for in the SL-chain of the environment (according to the visibility rules).

The list of formal parameters is dynamically compatible with the list of actual parameters if the following conditions hold:

- if Pfi is an input formal parameter, then Pai is defined and the dynamic type of Pfi is dynamically consistent with the type of the value of Pai, - if Pfi is an output/inout formal parameter, then Pai is defined and the dynamic type of Pai is statically consistent (!) with the dynamic type of Pfi, - if Pfi is a formal function (procedure), then the lists of formal parameters of Pfi and that of Pai must be of the same pattern (disregarding the descriptions of subprogram parameters). They may differ in the parameter identifiers, and they may differ in the class types of corresponding parameters (however, the class types must belong to the same prefix sequence), - if Pfi is a formal function, then the dynamic type of Pfi prefixes the dynamic type of Pai, or the two types are identical.

The above conditions are checked from left to right (i.e., for i=1, ..., k).

Recall that in the following description of prologue and epilogue the computations of the values and addresses for formal parameters and actual ones are performed in the syntactic environment of the object O1.

Prologue of an object.

The prologue consists of the following steps:

(i) The frame for a new object O is allocated, the object O1 is called the dynamic father of the object O. The sequence of dynamic fathers creates a chain called the DL chain (DL for dynamic links);

(ii) For the input and inout formal parameter Pf, the value of the corresponding actual parameter is computed and assigned to Pf;

(iii) For the output and inout formal parameter Pf, the address of the corresponding actual parameter Pa is computed (in other words, the prologue of the assignment of Pf to Pa is performed);

(iv) For the formal type parameter Pf, the corresponding actual type Pa is determined. According to 6.1.5. the valuation of the object O assigns the text of the determined type Pa to the identifier Pf. Therefore as long as that object exists the access to Pf is well-defined and connected with Pa;

(v) For the formal subprogram parameter, the actual subprogram is fixed (in the same way as the determination of the allocated unit and its environment).

After the execution of the epilogue the control is transferred to the object O. Let M1, ..., Mk=M be the prefix sequence of M. The execution of the statements from the object O begins from the first statement of the unit M1 (for the description of the further progress of computation, see inner statement, 9.1.3.). Note that those statements are executed in the syntactic environment of the object O. When the control returns to the calling object O1, the actions of the epilogue are performed.

Epilogue of an object.

The epilogue consists of the following steps:

(i) For the output or inout formal parameter Pf the actions of the epilogue for the assignment Pa:=Pf are performed, where Pa is the actual parameter corresponding to Pf. It means that the value of Pf (computed during the execution of the body) is assigned to Pa (this address was computed during the prologue);

(ii) If the unit is a function, then the result of the given call is determined by the current value of the corresponding variable result,

(iii) If the unit is a generalized class, then the result of a new M is a reference to the object O;

(iv) A terminated object (cf. 9.1.3.) of a block or a subprogram is deallocated. However, the terminated object of a generalized class is accessible as

long as there is a reference pointing to it (unless it is directly deallocated by means of the kill statement).

Remark. Note that for the input formal parameter Pf of non-primitive type, the value of the corresponding actual variable parameter Pa may be updated (both the formal parameter and the actual one point to the same object). In order to access the value of Pa without the possibility of its modification one can use the copying statement Pf:=copy(Pf) at the end of the unit body.

End of remark.

Array generation.

SYNTAX.

```
   <array generation>:

  ----> array -----> <variable > -----> ( -->|
                                              |
 |<---------------------------------------------|
 |
 |--> <arithmetic expression> --> : --> <arithmetic expression>--> ) -->
```

A declaration of a variable of an array type fixes the type of the array elements; bound pairs are fixed at the time of generation.

CONTEXT.

A statement array Y dim (l:u) is a WFF if:

- Y is a variable of the type (arrayof)<i>T, where i>0, T is a type identifier;

- l, u are WFFs and arithmetic expressions.

The above statement is considered to be an assignment of a reference (to a newly created object) on the variable Y.

SEMANTICS.

The following actions are performed:

- determine the address of variable Y; - compute the values l1, u1 of expressions l, u; - put l0, u0 truncations of l1, u1 respectively; - check the condition l0<=u0; - generate an array object and assign its address to Y.

The initial values of attributes (l0), ..., (u0) depend on their type of the form (arrayof)<i-1>T. The value of an array type variable may be changed by means of assignment, copying, and generation statements. The generation of an n-dimensional array consists of n steps. The first dimension is generated: e.g. array Y dim (l1:u1), next the second dimension: e.g. for i:=l1 to u1 do array Y(i) dim (li2:ui2) od and so on. Non-regular arrays can be generated in this way. 9.1.2.2.

Deallocation statement

SYNTAX.

```
           <object deallocation>:
```

```
----> kill ----> ( ----> <object expression> ----> ) --->
```

CONTEXT and SEMANTICS.

A statement kill(X) is a WFF if X is a well formed object expression of compound type. The statement kill(none) is always WFF and it is equivalent to the empty statement. The statement is defined if X points to an object O not belonging to the SL chain or DL chain of an active object. By an active object we mean the object containing the statement currently being executed (notice that in the case of parallelism there may co-exist several active objects). The execution of the statement results in the deallocation of object O, all variables pointing to O are set to none. The deallocation of an object which belongs to the SL chain or DL chain of an active object results in a run-time error. The statement kill(X) where X points to a coroutine head is described in 9.1.4. The statement kill(X) where X points to a process is described in 11.1.

Remark.

After a block or subprogram termination, the corresponding object is automatically deallocated. On the other hand, the array, class, coroutine, or process objects are not automatically deallocated. The computer memory may be overloaded with such objects even if they are no longer referenced. Those objects are recovered with the help of the system program called the garbage collector. The user can help in the execution of that system program and increase the efficiency of his program execution if he deallocates unnecessary objects. One should realize, however, what are the effects of deallocation (in particular, a side effect consisting in the modification of the values of all variables which point to the same deallocated object).

End of remark.

Example.

The deallocation of a binary tree can be performed by means of the following recursive procedure:

```
unit tree_kill: procedure (n:node);
begin
  if n.l=/=none then call tree_kill(n.l) fi;
  if n.r=/=none then call tree_kill(n.r) fi ;
  kill(n)
end tree_kill
```

where the class node has the form

```
unit node:  class;
  var l, r: node ;
end node;
```

9.1.3.

9.1.3 Simple control statement

There are two kinds of simple control statements: a textual control statement and a dynamic control statement. In this section we shall consider the occur-

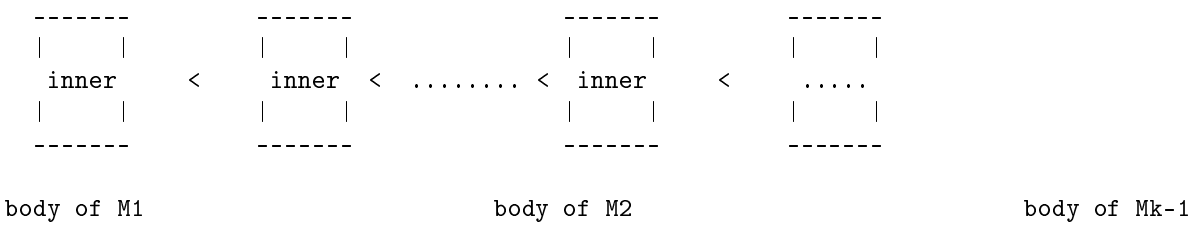
rence of a control statement in the object O of the unit M, in the body of the unit Mj, where M has the prefix sequence M1, ..., Mk=M, and 1<=j<=k.

SYNTAX.

```
    <simple control statement>:

-----> <textual control statement> -------->
   |                                    ^
   |--> <dynamic control statement> --->|

    <textual control statement>:

-------> inner ----->
 |                  |
 |                  |
 |-----> exit ----->|
 | |       |        |
 | |<------|        |
 |         |        |
 |---> repeat ----->|
```

SEMANTICS.

For j=1, ..., k-1 the execution of the inner statement results in the commencement of the execution of the unit Mj+1. The inner statement in the body of the unit Mk=M is empty.



The semantics of repeat and exit statements will be defined jointly with the semantics of a loop statement, see 9.2.3..

SYNTAX.

```
    <dynamic control statement>:

---------->  return ------------>
```

SEMANTICS.

In this section we shall describe the semantics of a return statement and a pseudo-statement end (which bound a unit declaration). An object may be

in one of the following three states: non-generated, generated, terminated. An object is non-generated until the control reaches the first return statement. From that moment an object becomes generated. An object is terminated after the execution of its end statement. The main program is considered to be always generated. A generated object is considered to have no dynamic father (its DL is none). Note that the execution of a terminated object cannot be resumed. However, the execution of the generated object of a coroutine or a process can be resumed and suspended. The return statement is empty if M is a coroutine and O is generated. If M is a block, subprogram, or generalized class and O is non-generated then O becomes generated. The control returns to the dynamic father of O. For a coroutine or process the statement following the return statement is a reactivation point.

Now we shall consider the execution of the final end. For j=2, ..., k the execution of the final end results in the control transfer to the statement following the inner statement from the unit Mj-1. Suppose that j=1. If O is a non-generated object of a coroutine, then O becomes generated and the control returns to the dynamic father of O. Otherwise (O is a coroutine/process object) the object O becomes terminated. The control transfer is the same as in the case of detach statement. Moreover, if M is a process, then the control becomes idle (and the corresponding processor may be released, see 11). 9.1.4.

9.1.4 Coroutine statement

SYNTAX.

```
     <coroutine  statement>:

------> detach ------------------------------------------------>
|                                                               ^
|-----> attach ----> ( ---> <object expression>--> ) -->|
```

CONTEXT and SEMANTICS.

By a chain of coroutine N with the head Ol we shall mean the DL chain of objects O1, ..., Ol such that: - for i=1, ..., l-1 the object Oi+1 is the dynamic father of Oi; - Ol is the generated object of the coroutine N; - Ol is non-terminated. Thus the chain contains non-generated objects with the exception of the head, which is generated but non-terminated. The execution of a kill(X) statement where X points to the head Ol of the coroutine chain results in a deallocation of the entire chain.

The chain may be in one of the following two states: - detached - the execution of the statements contained in this chain is suspended, the object O1 contains a distinguished point, called the reactivation point of the chain; - attached - a statement from the object O1 is currently executed.

In the case of a sequential program exactly one chain is operational, i.e., in the attached state. Note that a coroutine head may be the main program. Coroutine control statements change the states of coroutine chains. A reference to the coroutine chain W1 which has recently transferred the control to the chain W is associated with chain W. Let us denote this reference by CL (coroutine link). This link is then used by the detach statement. Suppose that the object

O (containing the occurrence of the coroutine control statement) belongs to the chain W of the coroutine N with the head Ol. The statement attach(X) is a WFF if X is a well formed object expression or the system variable main. The statement is defined if X points to the head O1 of a coroutine chain W1. The execution of the statement results in changing the state of W to a detached one and that of W1 to an attached one. The statement determined by the reactivation point of the chain W1 starts its execution. The CL link to the chain W is associated with the chain W1. If O=O1 then the statement is empty. The statement detach is defined except the case where the CL link of chain W is none. The execution of the statement results in detaching the chain W and attaching the chain W1 determined by the corresponding CL link associated with W. The statement following the detach statement is the reactivation point of the chain W. The execution of the chain W1 is resumed at its reactivation point.

9.2.

9.2 Compound statements

Compound statements enable iteration (loop statements) and case analysis (conditional and case statement).

SYNTAX.

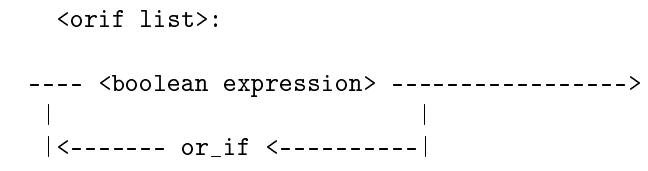
```
          <compound statement):

      ----------> <conditional statement> -------->
           |                                  ^
           |-----> <case statement> ------->|
           |                                  |
           |-----> <loop statement> ------->|
```

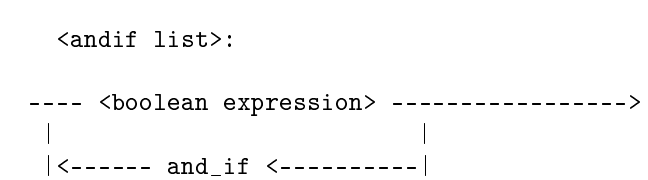
9.2.1.

9.2.1 Conditional statement

SYNTAX.

```
     <conditional statement>:

 ---> if --> <boolean expression> --> then --> <statement list>
          |                          |                   |
          |---> <orif list> ------->|                   |
          |                          |                   |
          |---> <andif list> ------>|                   |
                                                         |
                                                         |
                                                         |
        <-------------------------------------------------|
```





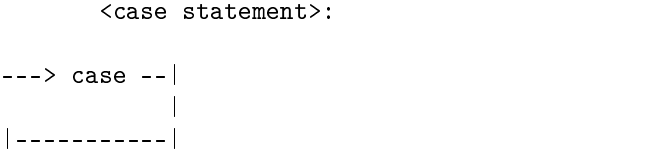
CONTEXT and SEMANTICS.
For the execution of an if statement of the form:

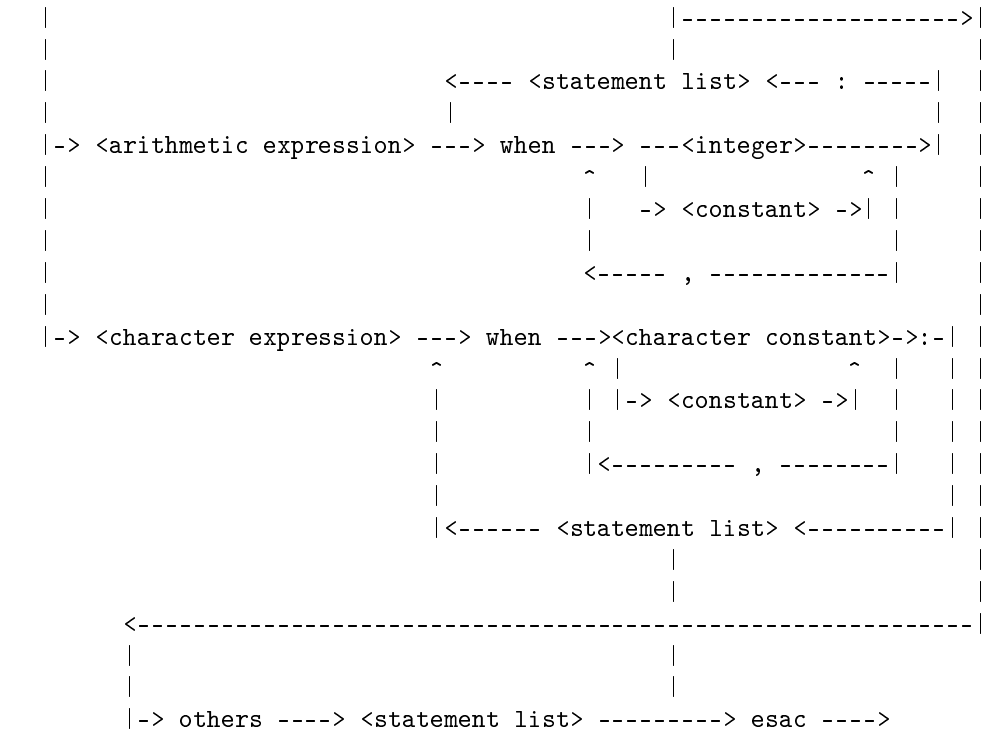
```
if  B1  or_if  B2 ... or_if Bj
then
  G
else
  H
fi
```

the boolean expressions B1, .., Bj are evaluated in succession until the first one evaluates to true, then the sequence G of statements is executed. If none of the boolean expressions evaluates to true, then the sequence H is executed. The conditional statement with the "else" part omitted is equivalent to the conditional statement with the empty statement following the else symbol. If the "andif" list occurs instead of the "orif" list, then the expressions B1, ..., Bj are evaluated in succession until the first one evaluates to false; then the sequence H is executed. Otherwise the sequence G is executed. When a boolean expression occurs instead of an "orif" or "andif" list, then its value controls the execution of the conditional statement in the standard manner. 9.2.2.

9.2.2 Case statement

SYNTAX.





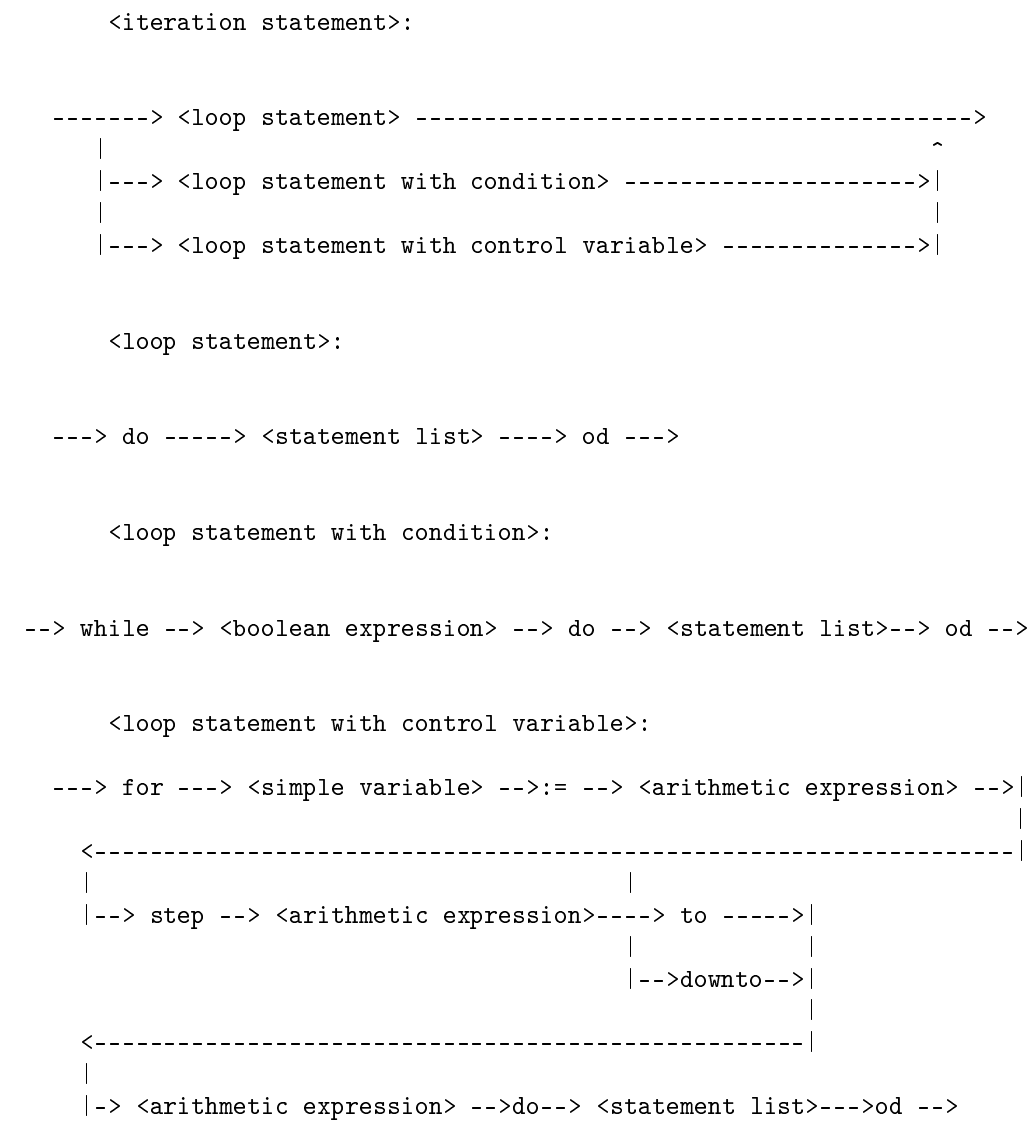
CONTEXT and SEMANTICS.
A statement:

```
case E
  when l1:G1
    ...
  when lk:Gk
  others H
esac
```

is a WFF if E is an arithmetic or character expression and l1, ..., lk are sequences of different constants. If the list H is empty, then the "others" part can be omitted. The case statement selects for execution a sequence Gi where the value of E belongs to the sequence li. The choice others covers all values (possibly none) not given in the previous choices. The choices in a case statement must be mutually disjoint and if the "others" part is not present they must exhaust all the possibile values of the expression E. 9.2.3.

9.2.3 Iteration statement

SYNTAX.



CONTEXT and SEMANTICS.

Let us start from the semantics of loop and exit statements. The loop statement:

```
do
   G
od
```

causes the iteration of the sequence G until an exit statement is encoutered. Consider the occurrence of the exit statement exit ... exit(k-times) where k >=1 . Let us denote this statement by H. Suppose that statement H occurs in l (l>=0) nested iteration statements G1, ..., Gl, i.e., the statement G2 is nested in

G1, G3 in G2, etc. Let M be the smallest unit enclosing that occurrence of H. If k>l then the execution of H causes the termination of the unit body M (jump to the final end). Otherwise the iteration statement Gk terminates, and either the execution of the iteration statement Gk-1 is continued if k<l or the execution of the outermost iteration statement G1 terminates if k=l. The keyword repeat may occur just after the sequence of exit's. Then the iteration statement Gk is continued (if k<=l), i.e., the control is switched not outside but to the beginning of the loop without the execution of the statements occurring after repeat. If the statement Gk is a loop statement with the while condition, then the consequtive iteration starts from the condition evaluation. If it is a for statement, then the consequtive iteration starts with the change of the controlled variable value.

Remark.

The goto statement is totally deleted from LOGLAN-82 (contrary to other languages, like ADA where goto within a single unit is allowed). The structured statements defined above were applied to many examples of known algorithms. These exercises showed that the proposed structured statements constitute a good balance point between a non structured goto-label statement and a classical while statement (which often requires auxiliary control boolean variables). Notice that the above unit M body is considered to be "non-concatenated", i.e., in the case of the jump to end symbol, this end is taken from the body of M, not from the body of M concatenated with its prefix sequence. We stress that the textual control statements do not lead outside one unit.

End of remark.

A loop statement with condition:

```
while  B
do
   G
od
```

is equivalent to a loop statement of the form:

```
do
   if not B then exit fi;
   G
od
```

A loop statements with controlled variables are of the forms:

```
  (*)  for i:=A1 step A2 to A3 do G od
 (**)  for i:=A1 step A2 downto A3 do G od
```

The controlled variable i must be of discrete type. The statement (*) is equivalent to the following sequence of statements:

```
Av1:=A1; Av2:=A2; Av3:=A3;  i:=Av1;
while Av3>=i
do
   G;
   i:=i+Av2
od
```

The statement (**) is equivalent to the above sequence of statements with the condition Av3>=i replaced by Av3<=i and the assignment i:=i+Av2 replaced by i:=i-Av2. The variables Av1, Av2, Av3 are fictitious variables introduced only in order to define the semantics. The expression step A2 may be omitted if the value of A2 equals 1. The value of the controlled variable i should not be modified inside the loop (however, the result of such a modification would be well-defined). Moreover, its value is determined when loop is terminated according to the introduced semantics.

Example 9.1. *A palindrome is a word which is identical when written from left to right and conversely. The given algorithm looks for the first occurrence of a palindrome in a text and writes its length, (if found).*

```
unit palindrome: procedure;
var i, j, k: integer,
      text: arrayof character;
begin
  read(j);
  array text dim (1:j);
  for k:=1 to j
  do
    read (text(k))
  od;
  for i:=2 to j
  do
    k:=1;
    while text(k)=text(i-k+1)
    do
       k:=k+1;
       if k>i-k+1
       then
         write ("found at i-th item");
         return
       fi
    od
  od;
  write ("not found")
end palindrome;
```

Example 9.2. *A dictionary is a data structure S with the following operations:*

$member(x, S)$ - determines whether x is a member of S,

insert(x, S) - *replaces* S *by the union of* $S \cup \{x\}$,

delete(x, S) - *replaces* S *by the difference of* $S \setminus \{x\}$.

The elements of the set S *are assumed to be of the same type* T *and to be ordered by the relation less. A dictionary will be implemented by means of binary search trees BST. The user has the access to the operations member, insert, and delete and does not have to bother about the way of their implementation. Below we propose how to accomplish these operations as coroutines.*

```
unit bst: class (type t; function less(x, y:t):boolean);
hidden  root, e, i, d;
var root: node, member: e, insert: i, delete: d;
  unit node: class (value: t);
  var l, r: node;
  end node;

  unit e: coroutine;           (*elem- output attribute*)
  close trick, q, v;
  var trick, elem: boolean, q, v: node, x:t;
  begin
    return;
    do trick, elem:=false;   (* loop for member *)
      q:=root;
      v:=none;
      while q=/=none
      do
        if less(x, q.value)
        then v:=q; q:=q.l
        else
          if less(q.value, x)
          then v:=q; q:=q.r
          else elem:=true; exit
          fi
        fi
      od;
      inner;   (* elem=true  iff x belongs to S *)
      detach;
    od
  end e;

  unit help: E coroutine;
  taken trick, elem, q, v, x;
  begin
    inner;  (* trick=true iff x does not belong to S *)
    if not trick then exit fi;
    if v=none
    then root:=q
    else
      if less(x, v.value)
      then v.l:=q
```

```
        else v.r:=q
        fi   (* after insert or delete *)
      fi   (* attach new node q to its father v *)
    end help;


    unit i:  help coroutine;
    taken trick, elem, q, x;
    begin
      trick:=true;
      if elem then exit fi;
      q:=new node(x)    (* insert is a dummy if x belongs to S *)
    end i;

    unit  d: help coroutine;
    taken elem, q;
    hidden close w, u, s;
    var w, u, s: node;
    begin   (* delete is a dummy if x does belong to S *)
      if not elem then exit fi;
      w:=q;
      if q.r=none
      then q:=q.l
      else
        if q.l=none
        then q:=q.r
        else u:=q.r;
          if u.l=none
          then u.l:=q.l; q:=u
          else
            do s:=u.l;
               if s.l=none then  exit fi;
               u:=s
            od;
            s.l.:=w.l; u.l:=s.r;
            s.r:=w.r; q:=s
          fi
        fi
      fi;
      kill(w)
    end d;

  begin
    member:=new e;   insert:=new i;  delete:=new d;
    inner;
    kill(member);  kill(insert);  kill(delete)
  end bst;

  pref bst(t, less) block
  taken  member, insert, delete;
```

```
var y:t;
  ...
begin
  ...
  member.x:=y;
  attach(member);
  if  member.elem then ... fi;
  ...
  insert.x:=y;
  attach(insert);
  ...
  delete.x:=y;
  attach(delete);
  ...
end;
```

10.

Chapter 10

Exception handling

This section defines the facilities for dealing with errors or other exceptional situations that may arise during program execution. An exception is an event that causes a suspention of normal program execution. The occurrence of an exception is expressed by raising a signal. Executing some actions in response to the arising of an exceptional situation, is called signal handling.

Signal names are introduced by signal specifications. Signals can be raised by raise statements, or alternatively, their raising is caused by an occurrence of a run-time error. When an exception arises, the control can be passed to a user-pointed handler associated with the raised signal. The principles of determining a handler that responds to the raised signals are presented in 10.3. 10.1

10.1 Signal specification

SYNTAX

```
   <signal specification>:

 ---> signal --> <signal name> ---> ( --> <formal par. list> --> ) -->; -->
                |                  |                                    ||
                |                  |----------------------------------->||
                |<---------------------- , --------------------------|
```

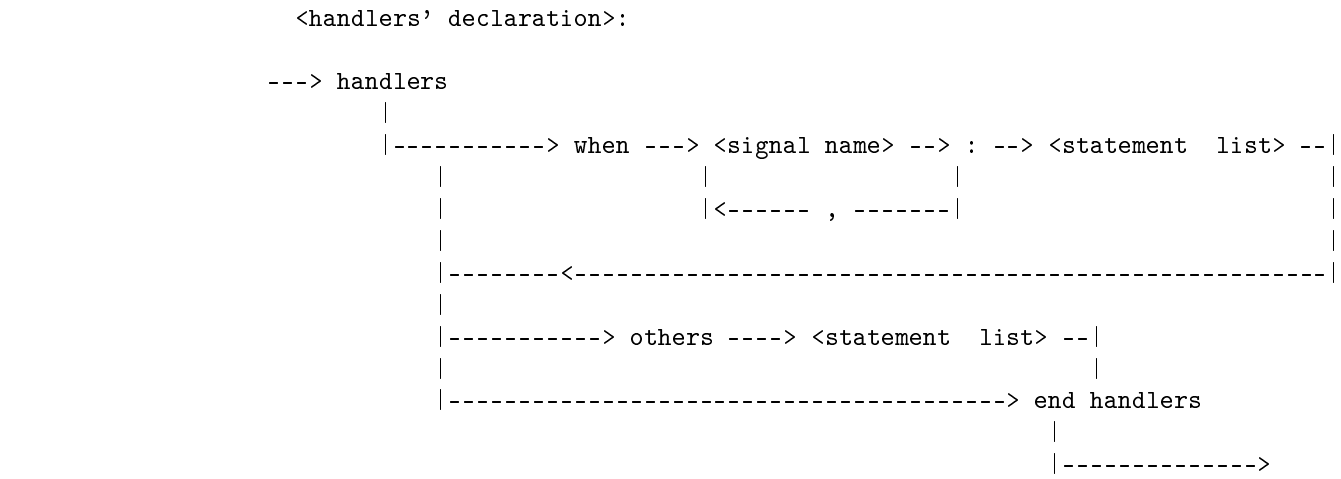
CONTEXT

The signal specification defines signals which can appear in raise statements and in signal handlers within the scope of the specification. The identifiers of system signals, i.e., signals associated with run-time errors, are not specified in the signal specification. Signal identifiers are not accessible by remote access. They can occur, however, in a hidden, close or taken list of a unit. 10.2

10.2 Signal handlers

The response to one or more signals is specified by a signal handler.

SYNTAX



CONTEXT

Handlers' declaration may appear at the end of the declaration part of a unit. All identifiers visible in the unit and the signal formal parameters may be used in the handler statements. A handler can handle the named signals. A handler corresponding to the choice others handles all signals not listed in the previously specified handlers, including those whose identifiers are not visible within the unit.

Any statement (except inner) whose occurrence in a unit is legal may occur in the handler.

Restrictions

The formal parameter lists of signals associated with the same handler must be identical.

Example

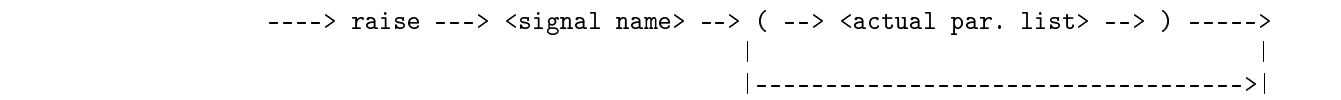
Example 10.1. `handlers`

```
    when emptytree: T:=new treelem; return;
    others write(" signal not handled"); raise Error;
  end handlers
```

10.3.

10.3 Signal raising

SYNTAX



CONTEXT

The signal name in the raise statement ought to be visible in the unit in which the raise statement appears. The formal and actual parameter lists of the signal must be compatible.

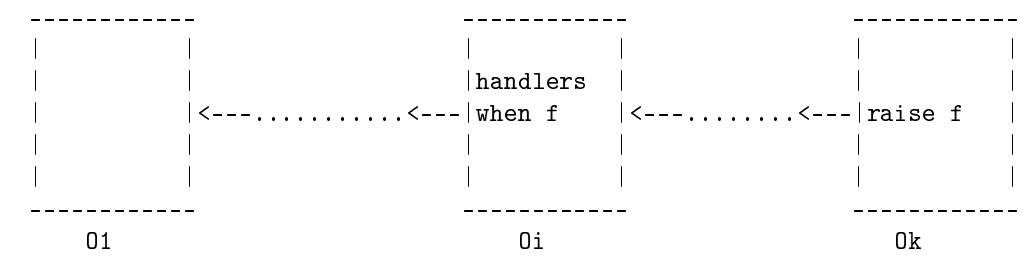
Example ——-

Example 10.2. *raise empty(exprstack); raise end_of_file (input);*

SEMANTICS

When a signal is raised, the normal process execution is suspended and the control is passed to a signal handler. The actual parameters are transmitted to the handler, as in the case of a procedure. However, the crucial point of exception handling is the way in which such a handler is searched for and activated. Below we present the principles of handler determination.

Let us assume that signal f is raised in object Ok. This object and its corresponding DL-chain may be illustrated as follows:



where O1 is the object of a coroutine or a process.

The objects in the DL-chain of Ok are successively checked whether they contain a handler for signal f or a handler corresponding to the choice others. The object Ok is checked first, next the object Ok-1 is checked and so on. This search stops when the first object Oi containing the handler for f or the handler for others is found. If such a handler is not found in this DL-chain, then the system trap handler is executed and the process terminates. For the situation presented in the figure, the handler from object Oi is executed, provided that none of the objetcs Oi+1, ..., Ok contains a handler for signal f or the handler for others.

In a concatenated object, i.e., in an object corresponding to a unit with a non-empty prefix, the handlers declared in the prefixing unit are covered by the handlers declared in the prefixed unit if they have the same identifiers. Thus the signal raised during the execution of the prefix statements may be handled by a handler declared in the prefixed unit. The handler corresponding to the choice others responds to all the signals not listed in the handlers declared in the units from the prefix sequence. The handler for the choice others is taken from the innermost unit (with respect to prefixing).

Example

Example 10.3.

```
block
  unit A: procedure;
  begin
    ...
    raise f
    ...
  end A;
  unit B: procedure;
  handlers
    when f: .....;        (* <----------- handler H1      *)
```

```
    end handlers
    begin
      ...
      call A;
      ...
      raise f;
      ...
    end B;
    signal f;
    handlers
      when f: .....;        (* <---------- handler H2     *)
    end handlers;
  begin
    ...
    raise f;
    ...
    call B;
    ...
  end
```

If signal f is raised in the block satement, handler H2 will be executed. If signal f is raised in procedure B or in procedure A, handler H1 will be executed.

```
block
  signal f;
  unit A:class;
    signal g;
    handers
      when g: .....;        (* <---------- handler G1    *)
    end handlers;
  begin
    ...
    raise f;
    ...
    raise g;
    ...
  end A;
  unit B:A class;
    handlers
      when f: .....;        (* <---------- handler F1    *)
      when g: .....;        (* <---------- hadller G2    *)
    end handlers;
  begin
    ...
    raise f;
    ...
    raise g;
    ...
  end B;
begin
```

```
   ...
end;
```

If signal f is raised in an object of class B, handler F1 will be executed. If signal g is raised in an object of class B, handler G2 will be executed even if the signal is raised in the statements of class A. 10.4.

10.4 Handler execution

A handler execution terminates if one of the special control statements is executed.

SYNTAX

```
<handler termination>:

     ------> return ----->|
     |                    |
--->|---> wind ---------------->
     |                    |
     |---> terminate ---->|
```

CONTEXT

The statements wind and terminate may appear only within a handler declaration. If none of them occurs in a handler statement list, the statement terminate is assumed to be the last statement in such a list. The execution of the statements wind and terminate causes an abnormal termination of the corresponding objects from the DL-chain (see below). In that case, the "last-will" statements are executed before the termination of the objects.

SYNTAX

```
<last-will statements>:

-----> last_will ----> : ---> <statement  list> ----------->
```

CONTEXT

Any unit body may be terminated with a sequence of statements labelled by last_will. They are not executed during normal program execution. The statement inner must not occur within the "last-will" statements.

SEMANTICS

Let us assume that a signal f raised in an object Ok is handled by a handler H from an object Oi:

```
  O1             Oi-1       Oi         Oi+1                  Ok
-----           -----     -----      -----                -----
|   | <---...<---|   |<---|   |<-----|   |<-----------<---|   |
-----  DL        ----- DL -----  DL -----  DL             -----
                           |                                    |
                           | SL                                 |
```

The statement return in a handler has a similar effect to that of the statement return in a procedure. The handler object is deallocated and the control is passed to the statement just following the corresponding raise.

The execution of the statement wind causes the termination and the deallocation of the objects H, Ok, ..., Oi+1. Before the termination of each of them, the "last-will" statements, if any, are executed. They complete the object execution. In the prefixed object the "last-will" statements of each prefix are successively executed, starting from the innermost and ending on the outermost prefix. When the termination and deallocation of these objects is completed, the control is passed to object Oi, where the computation is continued in a normal way. Note that the wind statement in the case of k=i is equivalent to return.

The statement terminate causes the termination and the deallocation of the objects H, Ok, ..., Oi+1, Oi. They are completed as in the case of wind, i.e., the "last-will" statements are executed as well. The control is passed to Oi-1, if such an object exists. If Oi-1 does not exists, i.e., Oi is the head of the DL-chain, then this head is terminated (cf. the semantics of the end statement of coroutine and process).

Remark

If any control statement (raise, detach, attach, etc.) is executed within the "last-will" statements and the control returns to the interrupted object, the execution of the "last-will" statements as well as the termination of the remaining objects in the DL-chain will be continued.

End of remark 10.5.

10.5 System signals

Some of the signals, called system signals, are predefined in the language. They are raised automatically when a run-time error or another exceptional system situation appears. System signals have no parameters. They are not declared in the signal specification, but the user may declare handlers for them. The execution of the statement return is not allowed in the handler responding to such a signal (note that sometimes the statement wind is equivallent to return).

The following signals are predefined in the language:

acc_error A remote access to a non-existing object or an error in the expression ...x qua A (x does not exist or the type of the object pointed to by x is not prefixed by the type A).

mem_error There is no free space for the allocation of a new object.

num_error A numerical error, such as for instance integer overflow, floating-point overflow, division by zero etc.

log_error Any kind of the LOGLAN Running System error (except access error) like e.g., an attempt to pass the control in a way inconsistent with the LOGLAN-82 rules, an attempt to kill an active object, etc.

con_error The value of an index expression exceeds the range of array indices or the array bounds are incorrect.

sys_error Any kind of system error like e.g., input-output error, parity error, etc.

Some other errors may also be introduced as system errors but are not predefined in the language.

11.

Chapter 11

Processes

Attention. This chapter need to be rewritten since the notion of process as implemented in the distributed version of Loglan82 differs from the first version; see User's guide and micro-manual.

Let us consider a snap-shot of a program's computation. This snap-shot is called a configuration. Up till now a configuration has consisted of a finite number of objects creating a number of coroutine chains. The main program is the only chain with the head of process type. Moreover, exactly one object has been considered "active" - i.e., its statements have been executed by a physical processor. By a physical processor we mean here an actual processor as well as the portion of time of a central unit. A configuration with many active objects illustrates the computation of a program with parallel statements. Concurrent computation to some extent generalizes coroutines, i.e., a configuration may contain many coroutine chains with heads of coroutine type and many process chains with heads of process type. The fundamental notion is that of a process. A process may be treated as a sequential program - only one statement of a process is being executed at a time. A parallel program consists of a number of processes. In LOGLAN-82 a process is a system type. A process object may be generated by means of the new statement. The generated process object is suspended with the execution of the return statement. This process may be resumed by means of the resume statement. After resumption, process statements are executed by a new processor, concurrently with the other active processes. During its computations, the process may suspend its actions (but not the actions of other processes) by means of the stop statement, then it may be resumed again, and so on. Observe that the attach and detach statements switch the processor from one object to another, while the resume and stop statements acquire and release a processor respectively. Resumption of a coroutine chain is connected with the control transfer from the active coroutine chain. Resumption of a process chain acquires new processor for that chain. Similarly, suspension of a coroutine chain gives the control back to another chain, while suspension of a process chain releases the processor. Note that a process object is more complex than a coroutine object. So coroutine operations are more efficient with respect to time and space.

In order to deal with communication among processes (e.g., by messages) as well as their competition in acquiring a resource (such as a shared variable) one should have the ability to define some synchronizing operations. Those

operations arise from the following constrains:

- timing, i.e., mutual exclusion of actions; - scheduling i.e., stating which of the waiting processes is to be resumed as the first one.

For this purpose some synchronizing facilities are provided. One may think of many such facilities, from low level ones, such as semaphores to high level ones, such as monitors. The decision which one of the synchronization methods should be chosen and incorporated into the language is weighty. The primitive tools are difficult to use, but they are efficient. The high-level constructs are safer, but they often limit parallelism (because of the strong synchronizing constraints).

13.

Chapter 12

File processing

13.1.

12.1 External and internal files

External files are named after character strings and denote peripheral devices or data sets. The logical and the physical structure of an external file depend on the given computer and its file system, and so, for the users of LOGLAN-82, external files are accessible via internal files only.

An internal file is an object of the predefined class type file. When an internal file is generated, it may be associated with an appropriate external file. Sometimes the user wish to generate an internal file not associated with any specified external one. Such a file is called a local file and its life-time is not longer than the life-time of the program where it has been generated.

A file is always treated as an unbounded sequence of bytes. A file can be read or written, and can be set to a required position. Each transmission from or on a file starts at the byte pointed out by the so-called current file position advanced by the number of transmitted bytes. File size is defined as the greatest number of a byte transmitted on the file.

There are some primitive facilities in the language which enable the user to read or write a specified number of bytes, to change the current file position, to obtain the file size, etc. All these facilities are in some sense low-level, since they operate on bytes. The user is able, however, to declare a class for file processing with high-level operations.

An example of a system class which defines a set of input-output operations applicable to files containing elements of a single type is shown in 13.6. Moreover this is not the only way to define high-level file processing. The user can declare, for instance, a class which defines operations applicable to files containing elements of mixed types, a class which defines operations on a file of arrays of various lengths, etc. 13.2.

12.2 File generation and deallocation

Before any operation on a file can be carried out, an internal file must be generated. If the user wishes to communicate with an external file, then the generated

internal file must be associated with that external one. When the generation of an internal file is in effect, the file is said to be open.

SYNTAX

```
<file declaration>:

-----> <variable list>  ----> :  file -------------->

<file generation>:

--> open
      |
      |
      (
      |
<object expression> ---> , ---> <string> ----> )  ------->
                      |                        |
                      |---------------------->|
```

SEMANTICS

Variables of file type are declared as any other variables of class type. An object of file type (internal file) has no attributes visible to the programmer. File generation differs from class generation. It is performed by an open statement. If the second argument appears, then a new internal file associated with an external one (identified by the string) is generated. The reference to such an internal file is set to the variable of type file occurring as the first argument. If there is only one argument, then a new local file is generated and the reference to the corresponding internal file is set to the variable of type file occurring as the argument of the operation. For instance:

```
open(X, "teletype")
```

generates a new internal file associated with the external file "teletype". Similarly

```
open(Y)
```

generates a new local file referenced by Y.

Thus the operation new is not applicable to files. Moreover, remote access to internal files is not permissible (no attributes visible to the user). Except file generation, remote access and prefixing, file type can be applied as any other class type. In particular, expressions of file type may be compared, assignments on variables of type file are allowed, the user can declare a function of type file, etc.

Remark

External file processing is not predefined in the language. The operations on external files, such as file creation, file deletion, file protection and so on, depend on the given file system. They may be introduced into the language as standard functions or procedures in the individual implementation.

End of remark

After processing has been completed on a file, it can be closed and the corresponding internal file may be deallocated. These actions are performed by the kill statement, where the argument points to the corresponding internal file.
13.3.

12.3 Binary input-output

SYNTAX

```
< binary  input-output  operations>:

--->  put ---> (---> <object expression>-> , ---> <expression list> --> ) ---->

--->  get ---> (---> <object expression>-> , ---> <expression list> --> ) ---->
```

SEMANTICS ———

Operation put transmits a sequence of bytes from memory to an open file (defined by the first parameter) at the current file position. This sequence of bytes is defined by the list of expressions. For any list element, going from left to right, the value of the expression is computed. If this value is primitive, then the transmitted bytes correspond exactly to the internal representation of the value. If the value is a reference to an object, then the transmitted bytes cover all non-system attributes of the referenced object. If this value is none, then no transmission is performed. Operation get transmits a sequence of bytes from an open file (defined by the first parameter) to the memory. If a list element is an object expression, then the transmitted bytes cover all non-system attributes of the referenced object (hence, if the value of this expression is none, then no transmission is performed). Otherwise, list element must be a variable of primitive type, and then the transmitted bytes cover exactly its internal representation. The sequence of transmitted bytes starts at the current file position.

For instance, let x be a real, i an integer and Y a reference variable to an object of type A:

```
unit A:class(j:integer);
var u, v, w:real;
end A;
```

Then the statement

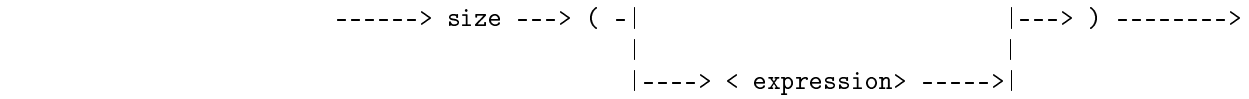
```
put(F, x, i, x+i, "nothing", Y)
```

transmits to file F the internal representation of the values of x, i, x+i, the internal representation of the text "nothing" (i.e., the sequence of characters) and the internal representation of the attributes j, u, v, w from the object referenced by Y. 13.4.

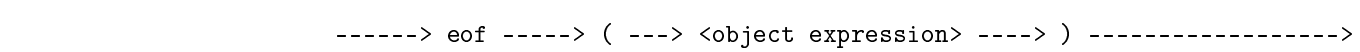
12.4 Other predefined operations

SYNTAX

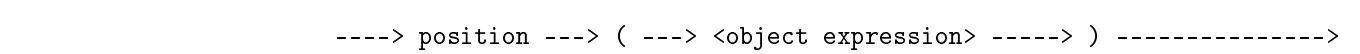
```
<size operator>:

                        |-----> <type> ----------->|
                        |                          |
```



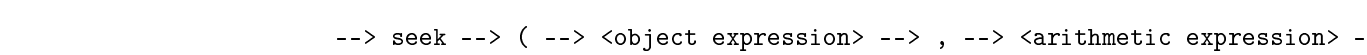
<eof operator>:



<position operator>:



<seek operation>:



SEMANTICS

The size operator of integer type gives the number of bytes of the internal representation of an argument. If the argument is an expression of primitive type, then the returned value may be computed at compilation time and equals the number of bytes of the internal representation of that primitive type. If the argument is an expression of class or array type, then the returned value gives the number of bytes of the object referenced by the argument (except system-attributes). If the object none is referenced, then the returned value is 0. Another kind of argument of size operator is type. It may be either an explicitly written type or a formal type. If the argument is a primitive type or a class type, then its length in bytes computed at compilation time is returned. If the argument is an array type, then its size cannot be established and so the expression is incorrect. If the argument is a formal type, the returned value is defined similarly but computed at run time. Thus when the actual type is array the run time error is raised. In all these cases size operator informs the user about the length in bytes of the internal representation of the argument (if possible). In particular, the argument may be a file and then the length in bytes of the corresponding external or local file is returned.

The argument of the boolean operator eof must be a file. It returns the value true iff the current position of the file exceeds or equals its size. The argument of the integer operator position must also be a file. It returns the current position of the file. The first argument of the seek operation must be a file. Then the current position of this file is set to the value defined by the second argument of the operation. 13.5.

12.5 Text input-output

Besides binary input-output LOGLAN-82 provides text input-output operations also. The operations read and write are available for input and output in human readable form. Namely, operation read decodes a sequence of bytes into the internal representation of the corresponding value before the transmission is performed. Similarly operation write encodes the internal representation of a value into the corresponding sequence of bytes before transmission is performed.

SYNTAX.

SEMANTICS.

An input statement read(F, y1, ..., yk) is correct if F is a file and y1, ..., yk are variables of integer, real, or character type. File F is treated as a sequence of symbols. The execution of that statement causes the input data represented as the corresponding sequence of symbols from file F to be read, decoded and assigned to y1, ..., yk respectively. The input statement is defined if the assignments are defined (going from left to right). An output statement write(F, E:A1) is correct if F is a file, E is an expression of a primitive type, and A1 is an arithmetic expression of integer type. Consider first the case where expression E is of integer type. The value of expression A1 determines the number of symbols to be outputed on file F. If the specified number of symbols is greater (less) than the number of symbols required for the representation of the value of expression E, then the value of E is preceded by the appropriate number of blanks (then the format indicates a standard one (dependent on an individual implementation). If expression E is of real type, then the output statement may be of the form write(F, E:A1:A2), here A1 and A2 are arithmetic expressions of integer type. The meaning of the expression A1 is hat described above, the value of the expression A2 determines the number of digits following the decimal point. In case of an output statement of the form write(F, E:A1), where E is of real type, the exponent part is always present. The absence of format indicates a standard one (dependent on an individual implementation). An output statement of the form write(F, E) where E is an expression of character type causes the external representation of E to be outputed on file F. If E is an expression of string type, then its external representation is outputed on file F. In this ase format A1 may appear and defines the maximal number of symbols which may be outputed,

i.e., if the length of a string exceeds the defined format, then the last symbols are dropped. In the statement write(F, E:A1:A2) format A2 is computed first (if present), format A1 is computed next (if resent), and finally the value of E is computed and outputed according to the defined formats. The execution of an output statement with a list results in the successive evaluations of the expressions A2, A1, E, and in the output of the computed value. Statement writeln outputs the end of line symbol after output is completed. If writeln has only the file parameter, then the end of the line symbol is outputed on file F. If no file is specified, a default standard input or standard output file is used. At the beginning of program execution, these files are open and associated with two implementation defined external files. 13.6.

12.6 Example of high-level file processing

A class defining high-level file processing is presented below. The user can prefix the main block of his program by such a class, and then, the high-level file operations are provided automatically.

```
unit input_output class;
hidden uni_file;
  unit uni_file :class(type T);
    hidden element_size;
    var F:file, element_size:integer;
    unit set_position:procedure(i:integer);
    begin
       call seek(F, i*element_size)
    end set_position;
    unit file_position:function:integer;
    begin
       result:=position(F) div element_size
    end file_position;
    unit end_of_file:function:boolean;
    begin
       result:=eof(F)
    end end_of_file;
    unit file_size:function:integer;
    begin
       result:=size(F) div element_size
    end file_size;
    unit read_element:procedure(output x:T);
    begin
       get(F, x)
    end read_element;
    unit write_element:procedure(x:T);
    begin
       put(F, x)
    end write_element;
  begin
     element_size:=size(T)
  end uni_file;
```

```
  unit inout_file:uni_file class(S:string);
  hidden F;
  begin
    open(F, S)
  end inout_file;
  unit in_file:inout_file class;
  hidden write_element;
  end in_file;
  unit out_file:inout_file class;
  hidden read_element;
  end out_file;
  unit local_file:uni_file class;
  hiddden F;
  begin
    open(F)
  end local_file;
  unit close_file:procedure(E:uni_file);
  begin
     kill(E.F); kill(E)
  end close_file;
end input_output;
```

Bibliography

[1] W. Bartol and D.Szczepańska. Data struture for simulation in loglan. Technical Report 373, IPI PAN, 1979.

[2] W. Bartol, A. Kreczmar, A. Litwiniuk, and H. Oktaba. Semantics and implementation of prefixing at many levels. In A. Salwicki, editor, *Proc. Logic of Programs and Their Applications*, volume 148 of *LNCS*, pages 45–80. Springer Verlag, 1983.

[3] O. J. Dahl, B. Myhrhaug, and K. Nygaard. Common base language (simula67). Technical Report S-22, NCC, Oslo, 1970.

[4] G. Mirkowska and A. Salwicki. Problems and theories inspired by the loglan project. In *Algorithmic Logic*, pages 298–347. PWN & D.Reidel, 1987.

[5] W. N. The programming language pascal. *Acta Informatica*, 1:35–63, 1971.

[6] A. Salwicki, W. Bartol, H. Oktaba, and T. Müldner. Loglan 77 - definicja języka programownia. Technical Report 20, Instytut Maszyn Matematycznych MERA, Warszawa, 1977.