DRAFT

Euclid Report

by B. W. Lampson[1], J. J. Horning[2], R. L. London[3], J. G. Mitchell[1], and G. J. Popek[4]

April 20, 1976

This document describes the Euclid language, intended for the expression of system programs which are to be verified.

Comments are earnestly solicited, and may be addressed to any of the authors. To facilitate comparison with the Pascal report, material in this report which is new is printed in large type, that which is copied from the Pascal report is in normal type, and material in the Pascal report which is omitted from the Euclid report appears here in small type.

Authors' addresses and support:

1. Xerox Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304

2. Computer Systems Research Group, University of Toronto, Toronto, Canada M5S 1A4
Supported in part by a Research Leave Grant from the University of Toronto.

3. USC Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291
Supported by the Advanced Research Projects Agency under contract DAHC-15-72-C-0308.

4. 3532 Boelter Hall, Computer Science Department, University of California, Los Angeles, CA 90024
Supported in part by the Advanced Research Projects Agency under contract DAHC-73-C-0368.

The views expressed are those of the authors.

This draft is circulated for criticism only. It should not be widely reproduced or quoted. A later version will be published.

DRAFT

Preface

This report describes a new language called Euclid, intended for the expression of system programs which are to be verified. Euclid draws heavily on Pascal for its structure and many of its features. In order to reflect this relationship as clearly as possible, the Euclid report has been written as a heavily edited version of the revised Pascal report.

Obviously, we are greatly indebted to Wirth, both for the aspects of the language which are copied from Pascal, and for the structure and much of the wording of the report. In addition, we have drawn on much other work in the design of programming languages, and in program verification.

1. Introduction

The development of the language Pascal is based on two principal aims. The first is to make available a language suitable to teach programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language. The second is to develop implementations of this language which are both reliable and efficient on presently available computers.

The desire for a new language for the purpose of teaching programming is due to my dissatisfaction with the presently used major languages whose features and constructs too often cannot be explained logically and convincingly and which too often defy systematic reasoning. Along with this dissatisfaction goes my conviction that the language in which the student is taught to express his ideas profoundly influences his habits of thought and invention, and that the disorder governing these languages directly imposes itself onto the programming style of the students.

There is of course plenty of reason to be cautious with the introduction of yet another programming language, and the objection against teaching programming in a language which is not widely used and accepted has undoubtedly some justification, at least based on short term commercial reasoning. However, the choice of a language for teaching based on its widespread acceptance and availability, together with the fact that the language most widely taught is thereafter going to be the one most widely used, forms the safest recipe for stagnation in a subject of such profound pedagogical influence. I consider it therefore well worth-while to make an effort to break this vicious circle.

Of course a new language should not be developed just for the sake of novelty; existing languages should be used as a basis for development whereever they meet the criteria mentioned and do not impede a systematic structure. In that sense, Algol 60 was used as a basis for Pascal, since it meets the demands with respect to teaching to a much higher degree than any other standard language. Thus the principles of structuring, and in fact the form of expressions, are copied from Algol 60. It was, however, not deemed appropriate to adopt Algol 60 as a subset of Pascal; certain construction principles, particularly those of declarations, would have been incompatible with those allowing a natural and convenient representation of the additional features of Pascal.

The main extensions relative to Algol 60 lie in the domain of data structuring facilities, since their lack in Algol 60 was considered as the prime cause for its relatively narrow range of applicability. The introduction of record and file structures should make it possible to solve commercial type problems with Pascal, or at least to employ it successfully to demonstrate such problems in a programming course.

The language *Euclid* has been designed to facilitate the construction of verifiable system programs. By a *verifiable* program we mean one written in such a way that existing formal techniques for proving certain properties of programs can be readily applied; the formal proofs might be either manual or

automatic, and we believe that similar considerations apply in both cases. By *system* we mean that the programs of interest are part of the basic software of the machine on which they run; such a program might be an operating system kernel, the core of a data base management system, or a compiler.

An important consequence of this goal is that Euclid is not intended to be a general-purpose programming language. Furthermore, its design does not specifically address the problems of constructing very large programs; we believe most of the programs written in Euclid will be smaller than about 2000 lines. While there is some experience suggesting that verifiability supports other desired goals, we assume the user is willing, if necessary, to obtain verifiability by giving up some run-time efficiency, and by tolerating some inconvenience in the writing of his programs.

In developing Euclid, we have sought to minimize deviations from the Pascal base; existing features of Pascal have been left unaltered unless there was a reason for change. We see Euclid as a (perhaps somewhat eccentric) step along one of the main lines of current programming language development: transferring more and more of the work of producing a correct program, and verifying its correctness, from the programmer and the verifier (human or mechanical) to the language and its compiler.

The main changes relative to Pascal take the form of restrictions, which allow stronger statements about the properties of the program to be made from the rather superficial, but quite reliable, analysis which the compiler can perform. In some cases new constructions have been introduced, whose meaning can be explained by expanding them in terms of existing Pascal constructions. The reason for this is that the expansion would be forbidden by the newly introduced restrictions, whereas the new construction is itself sufficiently restrictive in a different way.

The main differences between Euclid and Pascal are summarized in the following list:

> Visibility: Euclid provides explicit control over the visibility of identifiers, by requiring the program to list all the identifiers imported into a procedure or record, or exported from a record.
>
> Variables: The language guarantees that two identifiers in the same scope can never refer to the same or overlapping variables. There is a uniform mechanism for binding an identifier to a variable in a procedure call, on block entry (replacing the Pascal with statement), or in a variant record discrimination.
>
> Pointers: This idea is extended to pointers, by allowing dynamic variables to be assigned to collections, and guaranteeing that two pointers into different collections can never refer to the same variable.

Storage allocation: The program can control the allocation of storage for dynamic variables explicitly, in a way which confines the opportunity for making a type error as narrowly as possible. It is also possible to declare that some dynamic variables should be reference-counted, and automatically deallocated when no pointers to them remain.

Constants: Euclid defines a constant to be an identifier whose value is fixed throughout the scope in which it is declared.

Types: Types have been generalized to allow formal parameters, so that arrays can have bounds which are fixed only when they are created, and variant records can be handled in a type-safe manner. Records are generalized to include constant components, so they provide a facility for modularization.

For statement: A generator can be declared as a record type, and used in a for statement to enumerate a sequence of values.

Loopholes: features of the underlying machine can be accessed, and the type-checking can be overridden, in a controlled way. Except for the explicit loopholes, Euclid is believed to be type-safe.

Assertions: the syntax allows assertions to be supplied at convenient points.

Deletions: input-output, reals, multi-dimensional arrays, labels and gotos, and functions and procedures as parameters.

Other considerations in the design of Euclid are:

It is based on current knowledge of programming languages and compilers; concepts which are not fairly well understood, and features whose implementation is unclear, have been omitted.

Although the language is not intended for the writing of portable programs, it is necessary to have compilers which generate code for a number of different machines, including mini-computers.

The object code must be reasonably efficient, and the language must not require a highly optimizing compiler to achieve an acceptable level of efficiency in the object program.

Since the total size of a program is modest, separate compilation is not required (although it is certainly not ruled out).

2. Summary of the language

An algorithm or computer program consists of two essential parts, a description of *actions* which are to be performed, and a description of the *data* which are manipulated by these actions. Actions are described by *statements*, and data are described by *type definitions*. A data type definition essentially defines a set of values and the operations which may be performed on elements of that set.

The data are represented by *values*. A value may be *constant*, or it may be the value of a *variable*. A value occurring in a statement may be represented by a *literal* constant, an identifier which has been declared to be constant, an identifier which has been declared as a variable, or an *expression* containing values. Every constant or variable identifier occurring in a statement must be introduced by a constant or variable *declaration* which associates with it a data type, and either a value or a variable.

In general, a *definition* specifies a fixed value, type, procedure or function, and a *declaration* introduces an identifier and associates some properties with it. A data type may in Euclid be either directly described in the constant or variable declaration, or it may be referenced by a type identifier, in which case this identifier must be described by an explicit *type declaration*.

A *constant declaration* associates an identifier with a value; the association cannot be changed within the scope of the declaration. If the value can be determined at compile-time, the constant is said to be *manifest*; the expression defining a manifest constant must contain only literal constants, other manifest constants, and built-in operations.

The simple data types are the *scalar* types. Their definition indicates an ordered set of values, i.e., introduces identifiers standing for each value in the set. Apart from the definable simple types, there exist four *standard basic types*: *Boolean, integer, char* and *StorageUnit*. The *real* type has been omitted. For each standard type, there is a way of writing literal constants of that type: True and False for Boolean, numbers for integers, and quotations for characters. Numbers and quotations are syntactically distinct from identifiers. The set of values of type char is the character set available on a particular installation.

A type may also be defined as a *subrange* of a simple type by indicating the smallest and the largest value of the subrange.

Structured types are defined by describing the types of their components and by indicating a *structuring method*. The various structuring methods differ in the selection mechanism serving to select the components of a variable of the structured type. In Euclid, there are three basic structuring methods available: array structure, record structure, and set structure.

In an *array structure*, all components are of the same type. A component is selected by an array selector, or *computable index*, whose type is indicated in the array type declaration and which must be simple. It is usually a programmer-defined scalar type, or a subrange of the type integer. Given a value of the index type, an array selector yields a value of the component type. Every array variable can therefore be regarded as a mapping of the index

type onto the component type. The time needed for a selection does not depend on the value of the selector (index). The array structure is therefore called a random-access structure.

In a *record structure*, the components (called *fields*) are not necessarily of the same type. In order that the type of a selected component be evident from the program text (without executing the program), a record selector is not a computable value, but instead is an identifier uniquely denoting the component to be selected. These component identifiers are declared in the record type definition. Again, the time needed to access a selected component does not depend on the selector, and the record is therefore also a random-access structure. Records may include constant as well as variable components; manifest constant components, of course, occupy no storage. In particular, records may include procedures, functions and types as components. In this way, the operations which are defined on a data structure can be conveniently packaged with the structure.

Record components cannot be accessed outside the record body (which includes the bodies of procedure components) unless they are explicitly exported. Thus in a properly written program it is evident from the lexical structure how the state of a record can be altered.

A record type may be specified as consisting of several *variants*. This implies that different variables, although declared to be of the same type, may assume structures which differ in a certain manner. The difference may consist of a different number and different types of components. The variant which is assumed by the current value of a record variable is indicated by a constant of some scalar type which is called the *tag* field. Usually, the part common to all variants will consist of several components.

A *set structure* defines the set of values which is the powerset of its base type, i.e., the set of all subsets of values of the base type. The base type must be a simple type, and will usually be a programmer-defined scalar type or a subrange of the type integer.

A file structure is a sequence of components of the same type. A natural ordering of the components is defined through the sequence. At any instance, only one component is directly accessible. The other components are made accessible by progressing sequentially through the file. A file is generated by sequentially appending components at its end. Consequently, the file type definition does not determine the number of components.

Variables declared in explicit declarations are called *static*. The declaration associates an identifier with the variable, and the identifier is used to refer to the variable. The language guarantees that two identifiers which can legally be used in the same scope cannot refer to the same variable, or to overlapping variables.

In contrast, variables may be generated by an executable statement. Such a *dynamic* generation yields a *pointer* (a substitute for an explicit identifier) which subsequently serves to refer to the variable. This pointer may be assigned to other variables, namely variables of type pointer. Every pointer variable may assume values pointing to variables in a single *collection*, all of whose members are of the same type *T*, and it is said to be *bound* to this type *T*. It may, however, also assume the value **nil**, which points to no variable. Because pointer variables may also occur as components of structured variables, which are themselves dynamically generated, the use of pointers permits the representation of finite graphs in full generality. Although the language cannot guarantee in general that two pointer variables do not refer to the same variable, it can make this guarantee for two

pointers in different collections.

A *zone* can be associated with each collection to provide procedures for allocating and deallocating the storage required by variables in that collection; if the zone is omitted, a standard system zone is used. The program may free a dynamic variable explicitly, in which case the program is responsible for ensuring that no pointers remain to reference the non-existent variable. Alternatively, the collection may be *reference-counted*, in which case each variable is automatically freed when no pointers to it remain.

Throughout this report, then, the word *variable* means a container which can hold a value of a specific type. A variable may or may not be associated with an identifier. A *constant*, by contrast, is simply a value of a specific type. The fundamental difference is that assignment to a variable is possible.

A type declaration may have formal parameters; such a *parameterized* declaration represents a set of types, one of which is specified each time the type is referenced and actual parameters are supplied for the formals.

Two types are the same if their definitions are identical after any type identifiers that have been declared as *synonyms* have been replaced by their definitions, and any actual parameters and any identifiers declared outside the type have been replaced by their values.

The most fundamental statement is the *assignment statement*. It specifies that a newly computed value be assigned to a variable (or components of a variable). The value is obtained by evaluating an *expression*. Expressions consist of variables, constants, sets, operators and functions operating on the denoted quantities and producing new values. Variables, constants, and functions are either declared in the program or are standard entities. Euclid defines a fixed set of operators, each of which can be regarded as describing a mapping from the operand types into the result type. The set of operators is subdivided into groups of:

1. *arithmetic operators* of addition, subtraction, sign inversion, multiplication, division, and computing the remainder.

2. *Boolean operators* of negation, disjunction (or), and conjunction (and).

3. *set operators* of union, intersection, and set difference.

4. *relational operators* of equality, inequality, ordering, set membership and set inclusion. The results of relational operations are of type Boolean.

The *procedure statement* causes the execution of the designated procedure (see below). Assignment and procedure statements are the components or building blocks of *structured statements*, which specify sequential, selective, or repeated execution of their components. Sequential execution of statements is specified by the *compound statement*; conditional or selective execution by the *if statement*, and the *case statement*; and repeated execution by the *repeat statement*, the *while statement*, and the *for statement*. The if statement serves to make the execution of a statement dependent on the value of a Boolean expression, and the case statement allows for the selection among many statements according to the value of a selector. The discriminating case statement provides a safe way of discriminating the current variant of a variant record. The for statement is used

when a bound on the number of iterations is known beforehand, and the repeat and while statements are used otherwise.

A *block* can be used to associate a set of declarations with a statement. The identifiers thus declared have significance only within the block. Hence, the block is called the *scope* of these identifiers, and they are said to be *local* to the block. Since a block may appear as a statement, scopes may be nested. A type declaration also defines a scope in a similar way.

A block can be given a name (identifier), and be referenced through that identifier. The block is then called a *procedure*, and its declaration a *procedure declaration*. However, an identifier which is not local to a given procedure body is accessible in that body only if

> it is accessible in the immediately enclosing scope, and
>
> it is explicitly *imported* into the given procedure body.

Entities which are declared in the main program, i.e. not local to some procedure, are called global. A procedure has a fixed number of parameters, each of which is denoted within the procedure by an identifier called the *formal parameter*, which is local to the procedure body. Upon an activation of the procedure statement, an actual quantity has to be indicated for each parameter which can be referenced from within the procedure through the formal parameter. This quantity is called the *actual parameter*. There are two kinds of parameters: constant parameters and variable parameters; procedure, function and type parameters are not allowed. In the first case, the actual parameter is an expression which is evaluated once. The formal parameter represents a local constant whose value is the result of this evaluation. In the case of a variable parameter, the actual parameter is a variable and the formal parameter is bound to this variable. Possible indices or pointers are evaluated before execution of the procedure. In the case of procedure or function parameters, the actual parameter is a procedure or function identifier.

Functions are declared analogously to procedures. The only difference lies in the fact that a function yields a result, which may be of any type and must be specified in the function declaration. Functions may therefore be used as constituents of expressions. In order to eliminate side effects, assignments to non-local variables should be avoided within function declarations. Although Euclid does not forbid functions to have side effects, it is recommended that functions should not have side effects unless they are truly necessary. To this end, variable formal parameters, and imported variables, should be avoided within function declarations as much as possible.

Since Euclid is intended for the writing of programs which are to be verified (either mechanically or by hand), there are a number of explicit interactions between the language and the verifier, in addition to the many aspects of the language which have been motivated by the desire to ease verification. These explicit interactions fall into two main categories:

> embedding of assertions in the program: the special symbols **assert, invariant, pre** and **post** prefix assertions which are written as comments and ignored by the compiler, but presumably will be used by the verifier, which can take advantage of their relationship to the

structure of the program.

compiler-generated assertions: in cases where the compiler needs to be able to assume that some condition holds, but is unable to deduce that it does, the compiler may generate an assertion (in a new listing of the program) for the verifier, and then proceed as though confident of its truth. The legality of the program may then depend on the truth of the compiler-generated assertion.

3. Notation, terminology, and vocabulary

According to traditional Backus-Naur form, syntactic constructs are denoted by English words enclosed between the angular brackets < and > . These words also describe the nature or meaning of the construct, and are used in the accompanying description of semantics. Possible repetition of a construct is indicated by enclosing the construct within metabrackets { and } . The symbol <empty> denotes the null sequence of symbols.

The basic vocabulary of Euclid consists of basic symbols classified into letters, digits, and special symbols.

<letter> ::= 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 | 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

<octal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<digit> ::= <octal digit> | 8 | 9

<special symbol> ::=
+ | - | * | / | = | <> | < | > | <= | >= | (|) |
[|] | { | } | := | . | , | ; | : | ' | ↑ | **div** | **mod** |
nil | **in** | **or** | **and** | **not** | **if** | **then** | **else** | **elseif** | **fi** |
case | **of** | **discriminating** | **repeat** | **until** | **while** | **do** | **od** |
for | **decreasing** | **downto** | **begin** | **end** | **with** | **goto** | **const** | **var** |
type | **any** | **unknown** | **synonym** | **where** | **array** | **record** | **set** | **file** |
packed | **collection** | **reference-counted** | **forward** | **label** | **program** |
function | **procedure** | **inline** | **machine** | **code** | **unchecked** |
dependent | **aligned** | **at** | **word** | **bits** | **to** |
imports | **exports** | **pervasive** | **initially** | **finally** |
assert | **pre** | **post** | **invariant**

The construct

{ <any sequence of symbols not containing "}"> }

may be inserted between any two *tokens*: identifiers, literal constants (see 4), or special symbols. It is called a *comment* and may be removed from the program text without altering its meaning. The symbols { and } do not occur otherwise in the language, and when appearing in syntactic descriptions they are meta-symbols like | and ::= . The symbol pairs (* and *) are used as synonyms for { and }.

Throughout this report, various restrictions will be placed on *legal* Euclid programs. Many of these restrictions cannot be checked syntactically, and in some cases they involve dynamic conditions that are difficult (or impossible) to check statically. Nevertheless, programs that violate them are not considered to be meaningful Euclid programs. In general, it is the responsibility of the compiler to verify as many of these properties as it can, and to produce *legality assertions* for those that it cannot. Thus, any program whose legality assertions can all be verified is a legal Euclid program, with well-defined semantics. If **checked** is specified for a block (see 9.2.1), all legality assertions in the block are compiled into run-time checks, as an aid in detecting illegal programs, even before the verification process is complete.

3.1 Lexical structure

The text of a program is built up out of declarations and statements, collectively called *units*, according to the syntax specified below. In general units are separated by semi-colons. The syntax is constructed in such a way that a unit may *always* be legally followed by one or more semi-colons. In order to make it unnecessary to write semi-colons between units which appear on separate lines, a semi-colon is automatically inserted at the end of a line whenever the last token of the line is one of:

> identifier, literal constant,),], **nil**, **fi**, **od**, or **end** possibly followed by **if**, **do**, **case**, **record**, **code** or an identifier,

and the first token of the next line is one of:

> identifier, literal constant, **if**, **case**, **repeat**, **while**, **for**, **begin**, **const**, **var**, **type**, **function**, **procedure**, **machine**, **packed**, **program**, **pervasive**, **initially**, **finally**, **assert**, **pre**, **post**, or **invariant**.

Commas are used as separators in scalar types, element lists, parameter lists and identifier lists in declarations.

There are various kinds of brackets which are used to group declarations and statements for various purposes. The following list gives the possible closing brackets for each opening bracket.

if	**end** or **end if** or **fi**
do	**end** or **end do** or **od**
begin	**end**, or **end** <procedure or function identifier> if the block is the body of a procedure or function
case	**end** or **end case**
record	**end** or **end record** or **end** <record identifier> if the record declaration is the definition of a type identifier.
code	**end** or **end code** or **end** <identifier>

4. Identifiers, numbers and strings

Identifiers serve to denote constants, variables, types, procedures and functions. Their association must be unique within their scope of validity, i.e., within the block or type in which they are declared (see 6, 10 and 11).

<identifier> ::= <letter>{<letter or digit>}

<letter or digit> ::= <letter> | <digit>

The usual decimal notation is used for numbers, which are the literal constants of the data type integer (see 6.1.2.). The letter E preceding the scale factor is pronounced as "times 10 to the power of". Numbers may also be written in octal or hexadecimal notation.

<digit sequence> ::= <digit>{<digit>}

<unsigned integer> ::= <digit sequence>

<unsigned real> ::= <unsigned integer>.<digit sequence> |
<unsigned integer>.<digit sequence>E<scale factor> |
<unsigned integer> E <scale factor>

<unsigned number> ::= <unsigned integer> | <unsigned real>

<scale factor> ::= <unsigned integer> |
<sign><unsigned integer>

<sign> ::= + | -

<hex digit> ::= <digit> | A | B | C | D | E | F

<unsigned number> ::= <digit> {<digit>} |
<octal digit> {<octal digit>} #8 |
<hex digit> {<hex digit>} #16

Examples:

1 100 717#8 CAD1#16 123#16

Sequences of characters enclosed by quote marks are called strings. Strings consisting of a single character are the constants of the standard type char (see 6.1.2). Strings consisting of n (>1) enclosed characters are the constants of the type (see 6.2.1)

packed array [1..n] **of** char

Note: If the string is to contain a quote mark, then this quote mark is to be written twice.

<string> ::= '<character>{<character>}'

Sequences of characters enclosed by quote marks are called *literal string constants.* They are the literal constants of the standard type string (see 6.2.1). A character code which is not in the printing character set can be represented in a literal string constant in two ways:

*ddd where each *d* stands for an octal digit, represents the character code with the octal representation ddd;

*S, *T, *N, **, *', *" represent space, tab, newline, *, ' and " respectively.

A ' may also be represented by two successive ' characters.

<literal string> ::= ' { <extended character> } '

<extended character> ::= <character> | * <extension> | ' '

<extension> ::= <octal digit> <octal digit> <octal digit> | S | T | N |
* | ' | "

Examples:

' ' 'A' ';' '''' 'Here comes a null: *000 and there it went"
'Euclid' 'THIS IS A STRING' 'This*Sis*Sa*Sstring'

A single character preceded by a double quote is a literal constant of the standard type char (see 6.1.2). The * convention may also be used in these constants.

<literal char> ::= " <extended character>

Examples:

"a "*S {space character} "*000 {the NUL character}
"*" {a double quote character, *not* a string containing a single *}

5. Manifest constants

A manifest constant is an expression which is a synonym for a literal constant.

<manifest constant identifier> ::= <identifier>

<constant> ::= <unsigned number> | <sign><unsigned number> |
<constant identifier> | <sign><constant identifier> | <string>

<constant definition> ::= <identifier> = <constant>

<literal constant> ::= <unsigned number> | <literal string> |
<literal char> | <scalar value identifier>

<manifest constant> ::= <literal constant> |
<manifest constant identifier> |
<manifest constant expression>

6. Data type declarations

A data type determines the set of values which variables and constants of that type may assume and the set of basic operations that may be performed on them, and associates an identifier with the type. Parameterized types are introduced in 6.4. A type identifier declared as a *synonym* is considered to be the same type as its definition; otherwise the identifier is a different type.

A type declaration introduces a new scope in which the formal parameters of the type, if any, are declared (see 6.4). If the type definition is a record type, the new scope is closed (see 7.4), and identifiers defined outside are inaccessible unless imported. If the type definition is not a record type, however, the new scope is open and importing is not necessary.

An identifier must be delcared before it is used. When there are mutually recursive procedures or types, however, it is impossible to give the *definition* of every identifier before its use. In this situation, a definition of **forward** may be given instead, and later another declaration, of the form **type** *T*=... (or **procedure** *P*=...) must appear to provide the true definition.

<type> ::= <simple type> | <structured type> | <pointer type> |
<type identifier> | <parameterized type reference>

<type identifier> ::= <identifier>

<type declaration> ::= **type** <synonym> <type identifier>
<formal parameter list> = <type definition>

<synonym> ::= **synonym** | <empty>

<type definition> ::= <type> <where clause> | **forward**

<where clause> ::= **where** <formal parameter list> | <empty>

Examples:

type **synonym** *SameOldType* = *SomeType*
type *NewType* = *SomeType*

6.1. Simple types

<simple type> ::= <scalar type> | <subrange type>

6.1.1. Scalar types

A scalar type defines an ordered set of values by enumeration of the identifiers which denote these values.

<scalar type> ::= (<scalar value identifier> {,<scalar value identifier>})

<scalar value identifier> ::= <identifier>

Examples:

type *Color* = (*red, green, blue, orange, yellow, purple*)
type *Suit* = (*club, diamond, heart, spade*)
type *Day* = (*Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday*)
type *Sex* = (*female, male*)

Function components implicitly declared for each scalar, subrange (see 6.1.3) and index (see 6.2.1) type *T* (except real) are:

T.succ	the succeeding value (in the enumeration)
T.pred	the preceding value (in the enumeration)
T.max	the last value (in the enumeration)
T.min	the first value (in the enumeration)

If *x* is of such a type *T*, then *x*.succ and *x*.pred may be written as shorthand for *T*.succ(*x*) and *T*.pred(*x*).

For instance, *Suit.max* is *spade*, and *Day.min* is *Monday* (in both these cases, the dot notation is used to invoke a parameterless function associated with an object; e.g., *max* is a function associated with all objects whose types are simple, and, in particular, the *max* associated with all *Suit* values can be referred to as *Suit.max*).

6.1.2. Standard simple types

The following types are standard in Euclid, and are pervasive throughout the entire program:

integer The values are a subset of the whole numbers defined by individual implementations. Its values are the integers (see 4).

Its values are the positive and negative integers, in the mathematical sense. It is not possible to declare a variable to be of type integer. Instead, variables can be declared to be of some suitable subrange type.

Every implementation has two standard types, *signedInt* and *unsignedInt*. These are the largest subranges of integer type which can be represented in one machine word and which contain:

for signedInt, equal numbers of positive and negative numbers, or perhaps one more negative number.

for unsignedInt, 0 and no negative numbers.

An operation is called *well-behaved* if its operands are in the

range signedInt (unsignedInt) and it yields an integer result in the same range. Every implementation must support the evaluation of *any* expression in which all the operations are well-behaved (see 8.1). An implementation may also support the evaluation of expressions involving larger integers.

real — Its values are a subset of the real numbers depending on the particular implementation. The values are denoted by real numbers (see 4).

Boolean — Its values are the truth values denoted by the identifiers True and False.

char — Its values are a set of characters determined by particular implementations. They are denoted by the characters themselves preceded by a double-quote.

StorageUnit — Its values are undefined. This is the basic unit for storage allocation (see 6.3). There are no operations defined on this type.

6.1.3. Subrange types

A type may be defined as a subrange of another simple type by indication of the least and the largest value in the subrange. The first manifest constant specifies the lower bound, and must not be greater than the upper bound. If type A is a subrange of type B, and type B is a subrange of type C, we say that A is also a subrange of C. The succ, pred, max and min function components are defined for all subrange types.

<subrange type> ::= <manifest constant> .. <manifest constant>

Examples:

```
type oneToOneHundred = 1 .. 100
type symmetricRange = -10 .. +10
type Primary = red .. blue {the values of a Primary are red, green, or blue}
```

6.2. Structured types

A structured type is characterized by the type(s) of its components and by its structuring method. Moreover, a structured type definition may contain an indication of the preferred data representation. If a definition is prefixed with the symbol **packed**, this has no effect on the meaning of a program (although it may render an otherwise legal program illegal, if a component of the structure has been renamed as an entire variable; see 7.5), but is a hint to the compiler that storage should be economized even at the price of some loss in efficiency of access, and even if this may expand the code necessary for expressing access to components of the structure.

<structured type> ::= <unpacked structured type> |
packed <unpacked structured type>

<unpacked structured type> ::= <array type> | <record type> | <set type> |
<collection type> | <file type>

6.2.1. Array types

An array type is a structure consisting of a fixed number of components which are all of the same type, called the *component type*. The elements of the array are designated by indices, values belonging to the *index type*. The minimum element of an index type must not be greater than the maximum element. The array type definition specifies the component type as well as the index type.

<array type> ::= **array** (<index type> {,<index type>}) **of** <component type>
<index type> ::= <scalar type> | <constant> .. <constant>
<component type> ::= <type>

If n index types are specified, the array type is called n-dimensional, and a component is designated by n indices. Only one-dimensional arrays are allowed.

There are two standard components of an array type *T*:

T.IndexType	the index type
T.ComponentType	the component type

They resemble parameters of the type in the sense that if *a* is a variable of type *T*, then *a*.IndexType is the same as *T*.IndexType, and likewise for ComponentType.

Examples:

array (1 .. 100) **of** *signedInt*
array (-10 .. 10) **of** 0 .. 99
array (*Boolean*) **of** *Color*

6.2.2. Record types

A record type is a structure consisting of a fixed number of components, possibly of different types. The record type definition specifies for each component, called a *field*, its type and an identifier which denotes it. The scope of these *field identifiers* is the record definition itself, and if they are exported they are also accessible within a field designator (cf. 7.2) referring to a record variable of this type.

Record components may be constants, variables, procedures, functions or types, exactly like identifiers declared in procedures. Thus, a record serves as a package for a collection of related objects. Identifiers declared in a record are not known outside unless they are *exported* explicitly, so the packaging supplied by the record also provides protection against improper use of components which are intended to be known only within the record definition. The := and = operations (assignment and equality) must be exported explicitly; if they are not, assignment of records of the type, or comparison of two such records for equality, will not be allowed. It is always possible, however, to bind a record to a variable, or to use it as the definition of a constant. A record containing no procedures or functions automatically exports all of its identifiers, as well as := and =.

An exported identifier *x* is accessible (within a suitable field designator) in any scope in which the record type is accessible. When a type is exported, any

identifiers defined in the type and exported by its definition are also exported. This is relevant for record and scalar types; the latter export all their identifiers.

Any type may be exported. Note, however, that any identifier used free in a type (see 7.4) is treated like a parameter for the purpose of deciding whether two types are the same (see 6.5). Thus for many purposes any identifier used free in a type behaves like a formal parameter, whose corresponding actual parameter on every call is the value of the identifier in the enclosing scope at the time the type is referenced.

When a record definition appears in a type declaration, identifiers declared outside the record are not known inside unless they are known in the immediately enclosing scope, and either are pervasive or are explicitly imported into the record by the imports clause of the formal parameter list in the declaration (see 7.4). An identifier can only be imported into a record as a constant, never as a variable.

A record may include an *initial* statement which is executed whenever a new variable of the record type is created, and a *final* statement which is executed whenever such a variable is destroyed. It may also specify an *invariant* which is supposed to be true during the lifetime of the record variable (i.e. after the execution of the initial statement and before the execution of the final statment), except perhaps when one of the procedures or functions of the record has been called and has not yet returned. Like other assertions, this one is ignored by the compiler.

A record type may have several *variants*. In this case a constant of scalar type must be used as a selector in a **case** construction which enumerates the possible variants. This constant is called the *tag*, and its value indicates which variant is assumed by the record variable at a given time. Each variant structure is identified by a case label which is a manifest constant of the type of the tag field. Usually the tag will be a formal parameter of the type declaration in which the **case** appears (see 6.4).

There is a standard type string, declared as in the example below.

<record type> ::= **record** <exports clause> <record body> <end record>

<end record> ::= **end** | **end record** | **end** <identifier>

<exports clause> ::= **exports** (<export item> {, <export item>}) | <empty>

<export item> ::= <binding condition> <identifier> | := | =

<binding condition> ::= **const** | **var** | <empty>

<record body> ::= <field list> <initial action> <invariant> <final action>

<field list> ::= <fixed part> | <fixed part> ; <variant part> | <variant part>

<fixed part> ::= <record section> { ; <record section>}

<record section> ::= <field identifier> { , <field identifier>} : <type> | <empty>

<fixed part> ::= <declaration>

<declaration> ::= <pervasive> <declaration part >
{ ; <pervasive> <declaration part> }

<pervasive> ::= **pervasive** | <empty>

<declaration part> ::= <constant declaration part> |
<variable declaration> |
<procedure and function declaration> |
<type declaration>

<constant declaration part> ::= **const** <constant declaration>
{ ; <constant declaration> }

<constant declaration> ::= <id list> <type spec> = <constant definition>

<id list> ::= <identifier> { , <identifier> }

<type spec> ::= : <type definition> | <empty>

<constant definition> ::= <expression>

<variable declaration> ::= **var** <id list> <variable declarer> |
<machine-dependent variable declaration>

<variable declarer> ::= <type spec> <variable binding>
<variable initialization>

<variable binding> ::= == <variable> | <empty>

<variable initialization> ::= := <expression> | <empty>

<procedure or function declaration> ::= <procedure declaration> |
<function declaration>

<variant part> ::= **case** <tag field> <type identifier> **of**
<variant> { ; <variant>}

<variant> ::= <case label list> => (<record body>) | <empty>

<case label list> ::= <case label> { , <case label>}

<tag field> ::= <constant> | <empty>

<initial action> ::= **initially** <statement> | <empty>

<invariant> ::= **invariant** <assertion> | <empty>

<assertion> ::= <empty>

<final action> ::= **finally** <statement> | <empty>

Examples:

```
type Date = record
     Day: 1 .. 31;
```

```
        Month: (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
        Year: unsignedInt
        end record

type String(maxLength: 1 .. 256) = record
        length: 1 .. 256 := 0;
        text: array (1 .. maxLength) of Character
        end record

type Person(sex: Sex) = record
        name, firstname: String;
        age: 0 .. 200
        married: Boolean
        case sex of
              female => (pregnant: Boolean);
              male   => (enlisted: Boolean)
        end record

type Female = Person(female)

type Male = Person(male)
```

6.2.2.1 Machine-dependent records

A machine-dependent record type is a restricted kind of record type which allows the programmer to specify the exact position of each field. The compiler's responsibility is to check that fields do not overlap and that each field is at least large enough to hold values of its type. The size of values of machine-dependent records is determined by the largest word position specified in the declaration.

A machine-dependent record may have **const**, **procedure**, **function** and **type** components like an ordinary record. All its **var** components, however, must have position specifications. It may not have any parameters, or import anything except manifest constants, or have any variants.

An alignment clause in a machine-dependent record declaration forces a value of the record type to be allocated so that the machine address of its first word is divisible by some power of two.

```
<md record type> ::= machine dependent record <alignment clause>
                     <exports clause> <record body> <end record>

<alignment clause> ::= aligned mod <manifest constant> | <empty>

<machine-dependent variable declaration> ::=
                var <identifier>
                at word <manifest constant>,
                bits <manifest constant> to <manifest constant>
                <variable declarer>
```

Examples

```
type InterruptWord = machine dependent record aligned mod 8
      var device at word 0, bits 0 to 2: DeviceNumber;
      var channel at word 0, bits 3 to 5: 0 .. 7;
      var stopCode at word 0, bits 6 to 7: (finishedOk, errorStop,
      powerOff);
      var command at word 1, bits 0 to wordSize:
      ChannelCommand
      end record
```

6.2.3. Set types

A set type defines the range of values which is the powerset of its *base type.* Base types must be simple types. Operators applicable to all set types are:

- `+` union
- `-` set difference
- `*` intersection
- **in** membership

The set difference x-y is defined as the set of all elements of x which are not members of y.

<set type> ::= **set of** <base type>

<base type> ::= <simple type>

Examples:

```
type Hue = set of Color
type SubtractivePrimaries = set of red .. green
type SymSet = set of -5 .. +5
```

6.2.4. File types

A file type definition specifies a structure consisting of a sequence of components which are all of the same type. The number of components, called the length of the file, is not fixed by the file type definition. A file with 0 components is called empty.

<file type> ::= **file of** <type>

Files with component type char are called textfiles, and are a special case insofar as the component range of values must be considered as extended by a marker denoting the end of a line. This marker allows textfiles to be substructured into lines. The type text is a standard type predeclared as

```
type text = file of char
```

6.3. Pointer and collection types

Variables which are declared in a program (see 7.) are accessible by their identifier. They exist during the entire lifetime of the scope to which the variable is local, and these variables are therefore called *static* (or statically allocated). In contrast, variables may also be generated dynamically, i.e., without any correlation to the structure of the program. These *dynamic* variables are generated by the standard procedure new (see 10.1.2.); since they do not occur in an explicit variable declaration, they cannot be referred to by a name. Instead, access is achieved via a *pointer* value which is provided by new upon generation of the dynamic variable. A pointer type thus consists of an unbounded set of values pointing

to elements of the same type. No operations are defined on pointers except the test for equality, the pointer dereferencing operator ↑ which yields the variable referred to by the pointer, and the standard function Index which converts a pointer into an unsignedInt.

The pointer value **nil** belongs to every pointer type; it points to no element at all.

A dynamic variable must be an element of a *collection.* A collection is a variable which behaves very much like an array variable. Just as an element of an array variable *A* can be referenced by subscripting *A* with an index whose type is the index type, *A*.IndexType, of *A*, so an element of a collection *C* can be referenced by subscripting *C* with a pointer whose type is the pointer type, ↑*C*, of *C*. There are two differences:

> No two collections have the same pointer type. Hence the pointer alone is sufficient to specify the collection, and we allow *p*↑ as shorthand for *C*(*p*), where *p* is of type ↑*C*.
>
> There are no operations which produce pointer results, except the standard procedure new which creates a new variable. Hence the storage allocation strategy for collections can be quite different from the strategy for arrays.

The reason for having collections is that two pointers to different collections are guaranteed to point to different variables. Hence collections are a means by which the programmer can express some of his knowledge about the ways in which his program is using pointers. If he prefers not to do this, or has no knowledge about pointers to variables of type *T* which can be expressed in this way, he can simply declare a single collection of *T*s and use it everywhere.

There are no operations on collections. A collection may not be assigned to another collection. In fact, there is nothing to do with a collection except to subscript it, or to pass it as an actual parameter.

Associated with every collection is a *zone* which provides storage for its variables. A zone is a record with three special components (and possibly other components):

> a type StorageBlock which is a collection of a record type containing a special component (and possibly other components):
>
> Storage, an array of StorageUnit
>
> a function Allocate(unsignedInt) **returns** ↑ StorageBlock
>
> a procedure Deallocate(**var** ↑ StorageBlock)

These components need not be exported, since they are intended for use only by the standard procedures new and free (see 10.1.2).

A collection declared without a zone will get a standard system zone.

A collection can be *reference-counted*, in which case a variable in the collection will be freed automatically when no pointers to it remain. The optional manifest constant is an integer which gives the maximum reference count which should be maintained; any variable to which more than this number of pointers ever exists at one time may never be freed.

<collection type> ::= <counted> **collection of** <object type> <zone>
<pointer type> ::= ↑<collection>
<object type> ::= <type>
<counted> ::= **reference-counted** |
reference-counted <manifest constant> |
<empty>
<zone> ::= **in** <zone identifier> | <empty>
<zone identifier> ::= <identifier>

Examples:

type *Human* = **collection of** *Person*(**any**) **in** *EarthZone*
var *thePresident, aParent*: ↑*Human*

6.4 Parameterized types

It is possible to declare a parameterized type by including a formal parameter list in the type declaration:

type *T*(*a*: signedInt, *b*: *color*) = ...

or equivalently by writing a where clause at the end of the definition:

type *T* = ... **where** (*a*: *signedInt*, *b*: *color*)

The where clause is intended for declaring parameterized types in the formal parameter lists of procedures, as in

procedure *f*(*a*: **array** (0..*n*) **of** signedInt **where** (*n*: 1..1000), *b*: ...) = ...

Every reference to such a type, however, must have an actual parameter list which supplies values for all the formal parameters. The formal and actual parameters of a type are exactly like the formal and actual parameters of a procedure, except that a formal parameter of a type cannot be a variable.

When a parameterized type is referenced in the formal parameter list of a procedure, an actual parameter of the reference can be another formal parameter of the procedure (see 10.). Thus procedures can be written to accept actual parameters whose type is any reference to a parameterized type.

The syntax of type declarations allows a parameter to be used in one of the following ways:

as an array bound;

as the tag field of a variant record;

on the right-hand side in a constant declaration, and the constant might in turn be used in one of the above two ways;

as the collection name of a pointer;

as an actual parameter in a type;

as a constant in an initialization expression;

as a constant in an expression appearing in a statement (which could be in an initial action, a final action, or a procedure or function body).

The built-in type constructors **array** (*i..j*) **of** *T*, *i..j* (subrange), **case** *c* **of** ..., and ↑ *C* also take parameters; in fact, the first three can take parameters of any simple type, and the last can take any collection, unlike user-defined parameterized types, in which the types of the parameters are specified in the formal parameter list. For subrange the parameters must be manifest constants, so that a particular subrange type declaration in the program always produces exactly the same type. For **array** and **case** the parameters must be constants, but need not be manifest. Thus, textually identical occurrences of one of these constructors do not necessarily produce the same type. The **case** constructor is normally used in a type declaration in which its parameter is in turn declared to be a formal parameter of the declaration.

Parameters of a type may be referenced like record components; thus after

type *T*(*p*: *color*); **var** *x*: *T*(*red*)

the expression *x.p*=*red* is True. Unlike record components, type parameters are automatically accessible outside the type definition and need not be exported.

The special value **any** may be used as an actual parameter of a type reference, provided that the corresponding formal is of scalar type, and its only use is as the tag of a variant. Suppose *V* is such a parameterized type, with a formal parameter *s*, of scalar type *T*, used as a tag (there might be other formals, but they are omitted in this example). Then *V*(**any**) is a type whose values are the union of the values of *V*(*i*) as *i* ranges over all the elements of *T*. It differs from any particular *V*(*i*) in two important ways:

If *x* is declared to be of type *V*(**any**), only those components of *x* which are outside the case constructor with tag *s* can be referenced. A discriminating case statement (see 9.2.2.2) can be used to bind *x* to an identifier *y* whose type is *V*(*i*), and then all the components of *y* can be referenced in the scope of the discrimination.

The value of the parameter *x.s*, and hence the choice of variant, can be changed during execution by assignment to *x.s* (but not, of course, to *y.s* if *y* is of type *V*(*i*)). This is the only case in which any property of a variable which is determined by the parameters of its type can be changed after the variable has been created. Note that

when the type is changed in this way, any initializations specified for the variant are performed.

The special value **unknown** may be used as an actual parameter in a type reference, provided the reference appears as the object type of a collection. A variable in the collection can only be created by the standard procedure new, however (see 10.1.2), and when new is called , actual parameters must be supplied for all the **unknown**s in the object type. Hence a type never involves **unknown** except in the object type of a collection. When a pointer to **collection of** *T*(..., **unknown**, ...) is dereferenced to yield a variable *v*, that variable has type *T*(..., *x*, ...), where *x* is the value which was supplied to new when *v* was created. As in other cases where the parameters of types are not manifest constants, the compiler may have to generate legality assertions to ensure that the type of a dereferenced pointer has some property demanded by the context in which the dereferenced pointer is used. Note that any actual parameters in an object type other than **any** and **unknown** are evaluated when the collection is declared, *not* when a variable in the collection is created.

<parameterized type reference> ::=
    <type identifier> (<type actual parameter
    { , <type actual parameter> })

<type actual parameter> ::= <expression> | **any** | **unknown**

Examples of type definitions:

```
type FamilyMember(sex: Sex) = record
      Identity: Person(sex);
      Relations: record
            Mother, Father, Sibling: ↑Human;
            OldestChild: ↑Human
            end record
      end record;

type Family = record
      TheRoot: ↑ collection of FamilyMember
      end record
```

6.5 Type compatibility

Two types are the *same* if their *expanded definitions* are equal. The expanded definition of a type is obtained as follows:

start with the type;

replace each type identifier which was declared as a synonym by its definition, substituting the actual parameters for the formals;

do this repeatedly until there are no more identifiers;

the result is the expanded definition.

Two expanded definitions are equal if,

when all *extended parameters* of types (including array, subrange, case and ↑ constructors) are removed, they are identical sequences of basic symbols;

each extended parameter in one sequence is equal to the corresponding extended parameter in the other sequence.

The extended parameters of a type are the actual parameters, if any, together with the values of all identifiers used free in the type.

If the compiler cannot determine whether two types are the same, and they must be the same for the program to be legal, then the compiler will assume that they are the same, and generate a legality assertion guaranteeing this fact for the verifier to prove.

When a value is assigned to a variable, or a variable is bound to an identifier, the types must be compatible according to the following rules:

an operand for any operator other than dot, subscripting, and ↑, must have a type which is not parameterized;

in an assignment, both types must be the same, except that

ranges of variables on the left side may differ from the ranges of the corresponding components on the right side (note, however, that other parameters of types, such as array bounds, may *not* differ). In a legal Euclid program, each actual value being stored will be within the range of the corresponding variable. Where the compiler cannot verify the legality of an assignment, it will generate one or more legality assertions concerning the range of the actual value.

occurrences of **any** as an actual parameter in the type of the variable, and not within the object type of a pointer, may correspond to occurrences of any value in the other. Thus, a *T*(*red*) may be assigned to a *T*(**any**), but not the reverse. Furthermore, a pointer to *T*(*red*) may *not* be assigned to a pointer to *T*(**any**).

in a binding (see 7.5), the type T_v of the variable must be the same as the type T_i of the identifier. If the binding is part of a procedure or function call, however, actual parameters in the specification of T_i may be other formal parameters of the procedure or function (see 9.1.2).

The following table summarizes the transitions which are possible:

To (formal or left side)	*T*(*red*)	*T*(**any**)

From (actual or right side)		
T(*red*)	bind assign	assign
T(**any**)	discriminate	bind assign

7. Declarations and denotations of constants and variables

A constant declaration consists of an identifier denoting the new constant, followed optionally by its type, and then by an expression which defines its value. The defining expression is evaluated, and its value becomes the value of the constant, which can never change thereafter. The type of the constant, if specified, must be assignment-compatible with the type of the defining expression.

<constant> ::= <expression>

A variable declaration consists of a list of identifiers denoting the new variables, followed by their type and/or binding, and/or initialization. The binding, if present, specifies that the identifier (in this case there must be only one) is to refer to an already existing variable, rather than to a newly created one (see 7.5). The initialization is exactly equivalent to an assignment statement executed immediately after the declaration of which the variable declaration is a part. If the type is omitted, it is inferred from the binding or initialization.

The syntax for constant and variable declarations appears in section 6.2.2, since it is identical to the syntax for record components.

Every declaration of a file variable f with components of type T implies the additional declaration of a so-called buffer variable of type T. This buffer variable is denoted by f↑ and serves to append components to the file during generation, and to access the file during inspection (see 7.2.3. and 10.1.1.).

Examples:

const *i, j* = -1 {*i* and *j* will be *signedInt* and have the value -1}
const *typedConst*: *Color* = *red*

var *jimH, ralph, butler, jimM, gerald*: *Person*(*male*)
var *k, l*: -5 .. 5 := *i* {both variables initially have the value of *i*}
var *tableEntry*: *unsignedInt* == *table*(*j*) {*table*(-1) must be a valid reference and the type of *table*'s elements must be *unsignedInt*. *tableEntry* is simply another name for *table*(-1) over the scope of this declaration}
var *a* == *anotherVar* {*a* is *anotherVar* for the scope of this declaration}
var *a, b* := *i* {*a* and *b* are type *signedInt* and initially have the value -1}

Denotations of variables either designate an entire variable, a component of a variable, or a variable referenced by a pointer (see 6.3). Variables occurring in examples in subsequent chapters are assumed to be declared as indicated above.

Associated with every variable is a *main* variable which is entire; the variable is said to be *part* of its main variable. One variable is *part* of another if, roughly, an assignment to either can change the value of the other, and the space of possible values of the first variable is a (not necessarily proper) subset of the space of possible values of the second variable. The following sections define main variables and part precisely. "Part of" is a transitive relation: if *x* is part of *y* and *y* is part of *z* then *x* is part of *z*. It is also

reflexive: x is part of x. Two variables are the *same* if and only if each is part of the other. Two variables *overlap* if and only if one is part of the other.

<variable> ::= <entire variable> | <component variable> | <referenced variable>

7.1. Entire variables

An entire variable is denoted by its identifier, and is its own main variable. An entire variable is never part of another entire variable (see 7.5).

<entire variable> ::= <variable identifier>

<variable identifier> ::= <identifier>

7.2. Component variables

A component of a variable is denoted by the variable followed by a selector specifying the component. The form of the selector depends on the structuring type of the variable.

<component variable> ::= <indexed variable> | <field designator> | <file buffer>

<base variable> ::= <variable>

Corresponding to each kind of component variable described below, there is a corresponding constant expression which differs from the component variable in only one way: a constant record, array, collection or pointer appears in place of the base variable.

7.2.1. Indexed variables

A component of an n-dimensional array variable is denoted by the variable followed by an index expression. The main variable of the array variable is the main variable. The indexed variable is part of the array variable. An indexed variable $i1$ is part of another indexed variable $i2$ if and only if either they have the same array variable and the two indexes are equal, or the array variable of $i1$ is part of $i2$.

<indexed variable> ::= <array variable [<expression> {,<expression>}]

<indexed variable> ::= <array variable> (<expression>)

<array variable> ::= <base variable>

The type of the index expression must be the same as the index type declared in the definition of the array type.

Examples:

$a(12)$
$a(i+j)$
$b(red)$

7.2.2. Field designators

A component of a record variable, or a formal parameter of the type of any

variable, is denoted by the variable followed by the field identifier of the component or parameter. The field identifier or a record component must be exported in the type definition. A field designator is a variable only if the component was declared to be variable and exported as variable; otherwise it is a constant. If a field designator is a variable, its main variable is the main variable of the containing variable, and the field designator is part of the containing variable. A field designator *f*1 is part of another field designator *f*2 if and only if either their containing variables are the same and their field identifiers are identical, or *f*1's containing variable is part of *f*2.

<field designator> ::= <containing variable>.<field identifier>

<containing variable> ::= <base variable>

<field identifier> ::= <identifier>

Examples:

str.length
aPerson.age
son↑.father

7.2.3. File buffers

At any time, only the one component determined by the current file position (read/write head) is directly accessible. This component is called the current file component and is represented by the file's buffer variable.

<file buffer> ::= <file variable>↑

<file variable> ::= <variable>

7.3. Referenced variables

<referenced variable> ::= <collection variable> (<pointer variable>) | <pointer variable>↑

<collection variable> ::= <base variable>

<pointer variable> ::= <variable>

If *p* is a pointer variable whose collection *C* is of type *T*, *p* denotes that variable and its pointer value, whereas *p*↑ is short for *C*(*p*), which denotes the variable of type *T* referenced by *p*. The main variable of a referenced variable is the collection to which the variable belongs. Two referenced variables overlap if and only if their pointer variables are equal. A referenced variable is part of the collection variable.

Examples:

p1↑.father
p1↑.sibling↑.father

7.4 Scope rules

A scope is established by a type declaration or block, or a for or discriminating case statement. The scope extends from the **type**, **begin**, **for** or **case** to the end, except that it does not include any binding expressions in the variable declarations of the scope. A record type declaration, or a block which is the body of a procedure or function, is called a *closed* scope; other scopes are *open*. New identifiers are declared

as record components,

at the head of the block,

as parameters of a for or discriminating case, or

as formal parameters of a procedure, function or type.

These new identifiers are accessible within the newly established scope. Note that the name declared by a record type, procedure or function declaration is *not* declared in the closed scope which is the body, and must be imported explicitly into that scope if the definition is recursive.

An identifier used in a scope and not declared in that scope is said to be *free* in that scope. Any identifier which is free in a closed scope must either be declared **pervasive** in some enclosing scope, or be accessible in the immediately enclosing scope and explicitly imported into the closed scope

A new identifier may not be introduced which is the same as any other identifier accessible in the scope. Of course, an identifier accessible in the enclosing scope of a closed scope, and not imported, is not accessible, and hence may be reused.

An explicitly imported identifier has the same status as a newly declared one. The imports clause can specify for each identifier that it is imported as a variable, or as a constant (in which case it cannot be used as a variable, i.e. its value cannot be changed). An identifier can be imported as a variable only if is a variable in the enclosing scope. An identifier declared **pervasive** is automatically imported as a constant into all inner scopes, and that identifier may not be imported as a variable or redeclared in any inner scope.

A closed scope has the property that all its possible interactions with the rest of the world can be determined by examining its imports list, its parameters, and, in the case of a record, its exports list.

The definition for a constant, type, procedure or function declaration, and the initialization expression for a variable declaration, are within the scope of the block or record in which the declarations appear. Thus if this scope is closed, these expressions can contain only identifiers which are imported into the scope, occur as formal parameters to the procedure or record type, or are declared earlier in the scope.

7.5 Binding

An identifier may be *bound* to a variable when it appears

as a **var** formal parameter in a procedure or function declaration;

preceding an == in a **var** declaration in a block or a discriminating case statement.

We say that the variable is *renamed*. The scope of the binding is the scope of the declaration, and within this scope the identifier represents the variable. That is, the initial value of the identifier is the value of the renamed variable at the time of binding, and the last value assigned to the identifier will be the value of the renamed variable after control finally leaves the scope. If this variable is part of a component of an array, its index is evaluated when the scope is entered; if it is part of a referenced variable, the pointer variable is evaluated when the scope is entered.

The type and range of the identifier being bound must be the same as the type and range of the renamed variable to which it is bound (but see 10). A component of a packed structure must not appear as a renamed variable.

For open scopes (blocks and discriminating case statements), any variable free in the scope is considered to be renamed by the scope.

In order to allow a simple description of the rules for renaming variables, we will assume for the rest of this section that a procedure does not have any free variables (note that a record is already forbidden to do so). Any procedure which does have free variables is to be rewritten as a procedure which accepts the free variables as additional variable formal parameters, and every call is rewritten to supply the same variables as additional actual parameters. This also applies to procedures and functions in records: if a component of the record is a free variable in the procedure or function, that component is supplied as an additional actual parameter (in spite of the fact that it might not be exported). The rewritten program will behave exactly like the original one.

In order to ensure that the rewritten program is a legal one, however, we must (and do) impose the following requirement on the original program: any free variable in a procedure or function must have the property that it would be accessible as a variable in every scope which contains a call of the procedure if the field identifiers required to reach it were exported as variables.

The language ensures that an entire variable can never overlap (see 7.1) *any* other variable accessible in the same scope which has a different main variable, or in other words that

the value of an entire variable can change only

as the result of assignment to that variable or one of its parts, or

after exit from a procedure or function call in which that variable was the main variable of an actual parameter corresponding to a variable formal parameter;

> an assignment to an entire variable can never change the value of any other variable which is accessible in the scope containing the assignment, except one of its own parts.

To prevent binding from destroying this non-overlap property, the following restriction is imposed: no two variables which are renamed on entry to a scope can overlap. If the compiler cannot determine whether or not two variables overlap (e.g. *a(i)* and *a(j)* overlap iff *i=j*), it will assume that they don't, and generate a legality assertion to that effect for the verifier to deal with.

Since binding expressions are not part of the scope S in which they appear textually, but rather are in the enclosing scope, they may refer to identifiers which are not accessible in S, and may not refer to any identifiers declared in S.

Note that a pointer cannot be dereferenced within a given scope unless its collection is accessible in that scope, and cannot be dereferenced to a variable unless the collection is accessible as a variable in that scope; these rules are identical to the rules for indexed variables.

In general identifiers which are declared as constants cannot cause any aliasing problems, since their values can always be copied. Of course the compiler is free to use a pointer rather than copy the value if it can determine that the meaning of the program is the same; this will certainly be true if the variable involved does not overlap any variable accessible in the same scope. In other cases the value must be copied.

Copying will not work for collections, however, and it may be very inefficient for large arrays or records. Hence we impose a stronger rule for collections: if a collection is accessible in a scope as a constant, no variable which overlaps the collection can be renamed on entry to the scope. Furthermore, the same restriction is imposed on large arrays and records; the definition of "large" is implementation-dependent. If the programmer really wants a large array or record to be copied, he can declare a constant for that purpose.

A variable can be allocated to a specific address in memory by binding it to an element of the standard array Memory. This array is automatically declared in the outermost block of the program, but it is *not* pervasive and must be explicitly imported into any scope which references it. Note that a record type declaration, being a closed scope, will not be able to import Memory as a variable. This is not a defect in the language. The reason for naming a record type is so that multiple instances of the type can be conveniently declared, and it is not appropriate to create multiple instances of a record which imports Memory (or anything else) as a variable. It is perfectly all right to have

var *x*: **record** ... **end**

where the record can access Memory as a variable.

8. Expressions

Expressions are constructs denoting rules of computation for obtaining values of variables and generating new values by the application of operators. Expressions consist of operands, i.e., variables and constants, operators, and functions.

The rules of composition specify operator *precedences* according to six classes of operators. The multiplying operators have the highest precedence, then the adding operators, then the relational operators, then **not**, then **and**, and finally, with the lowest precedence, **or**. Sequences of operators of the same precedence are executed from left to right.

The elements of an expression are evaluated strictly from left to right, and all the elements are evaluated, except within expressions involving **and** and **or**; in these expressions the right operand is not evaluated if the left operand evaluates to False or True respectively.

The rules of precedence are reflected by the following syntax:

<unsigned constant> ::= <unsigned number> | <string> |
<constant identifier> | **nil**

<factor> ::= <variable> | <literal constant> | <function designator> | <set> |
(<expression>) | <adding operator> <factor> | **not** <factor>

<set> ::= <simple type identifier> | [<element list>]

<element list> ::= <element> {,<element>} | <empty>

<element> ::= <expression> | <expression>..<expression>

<term> ::= <factor> | <term><multiplying operator><factor>

<sum> ::= <term> | <sum><adding operator><term>

<relation> ::= <sum> | <sum><relational operator><sum>

<negation> ::= <relation> | **not** <relation>

<conjunction> ::= <negation> | <conjunction> **and** <negation>

<simple expression> ::= <term> |
<simple expression> <adding operator><term> |
<adding operator><term>

<expression> ::= <conjunction> |
<expression> **or** <conjunction>

Expressions which are members of a set must all be of the same type, which is the base type of the set. [] denotes the empty set, and [x..y] denotes the set of all values in the interval x...y.

Examples:

Factors:

x
15
(*x*+*y*+*z*)
abs(*x*+*y*)
[*red, c, green*] {where *c* is of type *Color*}

```
                    [1, 5, 10 .. 19, 23]
                    -x
```

Terms:

```
                    x*y
                    i div (1-i)
```

Sums:

```
                    x+y
                    hue1 + hue2
                    i*j+1
```

Relations:

```
                    x = 1.5
                    x not= 1.5
                    p <= q
                    (i<j) = (j<k)
                    c in hue1
                    c not in hue2
```

Negations:

```
                    not (p not= q)
                    not q
```

Conjunctions:

```
                    x<=y and y<z
                    p and not q
```

Expressions:

```
                    p or (x>y)
```

8.1. Operators

If both operands of the arithmetic operators of addition, subtraction and multiplication are of type integer (or a subrange thereof), then the result is of type integer. If one of the operands is of type real, then the result is also of type real. The compiler is expected to check that no overflow will occur during the evaluation of an expression; if it is unable to verify this, it must put out an assertion for the verifier to check.

8.1.1. The operator not

The operator **not** denotes negation of its Boolean operand.

8.1.2. Multiplying operators

<multiplying operator> ::= ***** | **/** | **div** | **mod** | **and**

operator	operation	type of operands	type of result
*	multiplication set intersection	real, integer any set type T	real, integer T
/	division	real, integer	real
div	division with truncation	integer	integer
mod	modulus	integer	integer

8.1.3. Adding operators

<adding operator> ::= + | - | **or**

operator	operation	type of operands	type of result
+	addition	integer, real	integer, real
	set union	any set type T	T
-	subtraction	integer, real	integer, real
	set difference	any set type T	T

When used as operators with one operand only, - denotes sign inversion, and + denotes the identity operation.

8.1.4. Relational operators

<relational operator> ::= = | <> | < | <= | >= | > | **in** | **not** <relational operator>

operator	type of operands	result
= <> < > <= >=	any scalar or subrange type	Boolean
in	any scalar or subrange type and its set type respectively	Boolean

Notice that all scalar types define *ordered* sets of values.

The operators <>, <=, >= stand for unequal, less or equal, and greater or equal respectively. The operators <= and >= may also be used for comparing values of set type, and then denote set inclusion. If p and q are Boolean expressions, p=q denotes their equivalence, and p<=q denotes implication of q by p. (Note that false < true).

The relational operators =, <>, <, <=, >, >= may also be used to compare (packed) arrays with components of type char (strings), and then denote alphabetical ordering according to the collating sequence of the underlying set of characters.

8.1.5 Other operators

operator	operation	type of operands	type of result
not	logical negation	Boolean	Boolean
and	logical "and"	Boolean	Boolean
or	logical "or"	Boolean	Boolean

8.2. Function designators

A function designator specifies the activation of a function. It consists of the identifier designating the function and a list of actual parameters. The parameters are variables and expressions, and are substituted for the corresponding formal parameters (cf. 9.1.2, 10, and 11).

<function designator> ::= <function identifier> |
<function identifier>(<actual parameter>

{,<actual parameter>})

<function identifier> ::= <identifier>

Examples:

Sum(*a*, 100)
GCD(147, *k*)
SumVectors(*a*, *b*)

9. Statements

Statements denote algorithmic actions, and are said to be *executable.* They may be prefixed by a label which can be referenced by goto statements.

<statement> ::= <unlabelled statement> | <label>:<unlabelled statement>

<unlabelled statement> ::= <simple statement> | <structured statement>

<label> ::= <unsigned integer>

9.1. Simple statements

A simple statement is a statement of which no part constitutes another statement. The empty statement consists of no symbols and denotes no action.

<simple statement> ::= <assignment statement> | <procedure statement> | <goto statement> | <escape statement> | <assert statement> | <empty statement>

<empty statement> ::= <empty>

9.1.1. Assignment statements

The assignment statement serves to replace the current value of a variable by a new value specified as an expression.

<assignment statement> ::= <variable> := <expression> | <function identifier> := <expression>

The variable (or the function) and the expression must be of the same type, with the following exceptions being permitted:

1. the type of the variable is real, and the type of the expression is integer or a subrange thereof.
2. the types of the expression and the variable are both subranges of the same type. If the value of the expression is not within the subrange of the variable's type, the program is illegal.
3. the type of the variable may have **any** as an actual parameter of a type where the type of the expression has some specific value (see 6.4).

Examples:

```
x := y+z
p := (1<=i) and (i<100)
hue := [blue, c.succ]
```

9.1.2. Procedure statements

A procedure statement serves to execute the procedure denoted by the procedure identifier. The procedure statement may contain a list of *actual parameters* which are assigned or bound to their corresponding *formal parameters* defined in the procedure declaration (cf. 10). The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively. There exist two kinds of parameters: *constant* parameters and *variable* parameters; procedure parameters (the actual parameter is a

procedure identifier), and function parameters (the actual parameter is a function identifier) are not permitted.

In the case of a *constant parameter*, the actual parameter must be an expression (of which a variable is a simple case). The corresponding formal parameter represents a local constant of the called procedure, and the current value of the expression is the value of this constant. In the case of a *variable parameter*, the actual parameter must be a variable, and the corresponding formal parameter is bound to this actual variable (see 7.5) during the entire execution of the procedure. A variable parameter must be used whenever the parameter represents a result of the procedure.

```
<procedure statement> ::= <procedure identifier> |
                          <procedure identifier> (actual parameter>
                          {,<actual parameter>})
<procedure identifier> ::= <identifier>
<actual parameter> ::= <expression> | <variable> |
                       <procedure identifier> | <function identifier>
```

Examples:

Random
Sort(*a*, *b*)

9.1.3. Goto statements

A goto statement serves to indicate that further processing should continue at another part of the program text, namely at the place of the label.

<goto statement> ::= **goto** <label>

The following restrictions hold concerning the applicability of labels:

1. The scope of a label is the procedure within which it is defined; it is therefore not possible to jump into a procedure.
2. Every label must be specified in a label declaration in the heading of the procedure in which the label marks a statement.

9.1.3 Escape statements

An escape statement serves to indicate that further processing should continue at the end of the smallest enclosing repetitive statement, or that control should return immediately from the function or procedure currently being executed. An expression must not appear in a **return** statement unless the statement is in a function body, and in that case the type of the expression must be assignment-compatible with the type of the function's result value.

A more elaborate escape construction such as Zahn's device can readily be simulated with a case statement, as the following example illustrates:

```
var flag : (a, b, finished) := finished
for ... do
    ...
```

```
        flag := a; exit;
        ...
        flag := b; exit;
        ...
        od
case flag of
        a => ...
        b => ...
        finished => ...
        end case
```

<escape statement> ::= **exit** | **return** | **return** <expression>

9.1.4 Assert statements

An assert statement introduces an assertion which is supposed to hold whenever control reaches that point in the program. The compiler treats it as a comment, as it does with the assertions supplied by invariant, pre and post clauses.

<assert statement> ::= **assert** <assertion>

9.2. Structured statements

Structured statements are constructs composed of other statements which have to be executed either in sequence (compound statement), conditionally (conditional statements), or repeatedly (repetitive statements).

<structured statement> ::= <compound statement> | <block> |
<conditional statement> | <repetitive statement> |
<with statement>

9.2.1. Compound statements and blocks

The compound statement specifies that its component statements are to be executed in the same sequence as they are written. The symbols **begin** and **end** act as statement brackets.

<compound statement> ::= <statement> {;<statement>}

Example: **begin** *z* := *x*; *x* := *y*; *y* := *z* **end**

A block is a compound statement within which new identifiers can be introduced. The symbols **begin** and **end** act as brackets to delimit the scope of the new identifiers. If **begin** is followed by **checked**, each legality assertion in the block is compiled into a runtime check, which aborts execution of the program if the assertion is false.

<block> ::= **begin** <checked> <declaration> ; <statement> **end**

<checked> ::= **checked** | <empty>

9.2.2. Conditional statements

A conditional statement selects for execution a single one of its component statements.

<conditional statement> ::= <if statement> | <case statement>

9.2.2.1. If statements

The if statement specifies that a statement be executed only if a certain condition (Boolean expression) is true. If it is false, then either no statement is to be executed, or the statement following the symbol **else** is to be executed.

The statement
if a **then** b **elseif** ... **fi**
is an abbreviation for
if a **then** b **else if** ... **fi fi.**

<if statement> ::= **if** <expression> **then** <statement> <elseif clause>
<else clause> <end if>
<elseif clause> ::= { **elseif** <expression> **then** <statement> }
<else clause> ::= **else** <statement> | <empty>
<end if> ::= **end** | **end if** | **fi**

The expression between the symbols **if** or **elseif** and **then** must be of type Boolean.

Note: The syntactic ambiguity arising from the construct

if <expression-1> **then if** <expression-2> **then** <statement-1>
else <statement-2>

is resolved by interpreting the construct as equivalent to

if <expression-1> **then**
begin if <expression-2> **then** <statement-1> **else** <statement-2>
end

Examples:

if *x*<1.5 **then** *z* := *x*+*y*; *c* := *blue*; **else** *c* := *red*; *z* := 0 **end if**

if *p1* **not**= **nil then** *p1* := *p1*↑.*father*; *p2* := **nil fi**

if *str.text*(1) = "$ **then** *country* := *UnitedStates*
elseif *str.text*(1) = "# **then** *country* := *GreatBritain*
else *country* := *Unknown*
fi

9.2.2.2. Case statements

The case statement consists of an expression (the selector) and a list of statements, each being labelled by a constant of the type of the selector. It specifies that the one statement be executed whose label is equal to the current value of the selector. A special label **otherwise** can be used to label a statement which should be executed if none of the other labels is equal to the current value of the selector.

If the selector is discriminating an object, each case list element of the case

statement becomes a new scope within which the identifier of the object is declared, either as a constant whose value is the expression in the object, or as a variable bound to the variable in the object. The expression or variable in the object must be a variant record, say of type *T*(**any**), and the tag of this record is used to select one of the case list elements. Within the element selected by a particular value of the tag, say *red*, the identifier has the type *T*(*red*). Thus

```
var anyx: T(any); ...;
case discriminating x=anyx of
   red => ...
   green => ....
   end case;
```

is more or less equivalent to

```
var anyx: T(any); ...;
case anyx.tag of
   red => begin const x: T(red)=anyx; ... end
   green => begin const x: T(green)=anyx; ... end
   ...
   end case;
```

except that the constant declarations in the latter would not be legal, because it is illegal to assign a *T*(**any**) to a *T*(*red*).

<case statement> ::= **case** <selector> **of** <case body> <end case>

<selector> ::= <expression> | **discriminating** <object>

<case body> ::= <case list element> {;<case list element>}
<otherwise element>

<case list element> ::= <case label list> => <statement> | <empty>

<otherwise element> ::= ; **otherwise** => <statement> | <empty>

<end case> ::= **end** | **end case**

<object> ::= <identifier> = <expression> |
var <identifier> == <variable>

Examples:

```
case operator of
      plus =>    x := x+y;
      minus =>   x := x-y;
      times =>   x := x*y
end

case i of
      1 => c := red;
      2 => c := blue;
      3 => c := green;
      4 => c := yellow
```

end case

9.2.3. Repetitive statements

Repetitive statements specify that certain statements are to be executed repeatedly. If a bound on the number of repetitions is known beforehand, i.e., before the repetitions are started, or if the repetition is controlled by a generator, the for statement is the appropriate construct; otherwise the while or repeat statement should be used.

<repetitive statement> ::= <while statement> | <repeat statement> | <for statement>

9.2.3.1. While statements

<while statement> ::= **while** <expression> <invariant> **do** <statement> <end do>

<end do> ::= **end** | **end do** | **od**

The expression controlling repetition must be of type Boolean. The statement is repeatedly executed until the expression becomes false. If its value is false at the beginning, the statement is not executed at all. The while statement

```
while B do S od
```

is equivalent to

```
if B then
   S;
   while B do S od
fi
```

Examples:

```
while a(i) not= x do i := i+1 od

while i>0 do
      if (i mod 2) not= 0 then z := z*x fi
      i := i div 2;
      x := x*x
      end do
```

9.2.3.2. Repeat statement

<repeat statement> ::= **repeat** <invariant> <statement> {;<statement>}
until <expression>

The expression controlling repetition must be of type Boolean. The sequence of statement between the symbols **repeat** and **until** is repeatedly executed (and at least once) until the expression becomes true. The repeat statement

```
repeat S until B
```

is equivalent to

```
begin S;
   if not B then
      repeat S until B
   fi
```

end

Examples:

```
repeat k := i mod j;
       i := j;
       j := k
until j = 0

repeat P(f↑); get(f) until NoMoreChars(f)
```

9.2.3.3. For statements

The for statement indicates that a statement is to be repeatedly executed while a progression of values is assigned to a new constant identifier called the *parameter* of the for statement.

<for statement> ::= **for** <parameter> <decreasing> **in** <generator> <invariant> **do** <statement> <end do>

<for list> ::= <initial value> **to** <final value> |
<initial value> **downto** <final value> |
<generator>

<parameter> ::= <identifier>

<initial value> ::= <expression>

<final value> ::= <expression>

<decreasing> ::= **decreasing** | <empty>

<generator> ::= <record type> | <index type>

The control variable, the initial value, and the final value must be of the same scalar type (or subrange thereof), and must not be altered by the repeated statement. They cannot be of type real.

The body of the for statement is a new scope within which the parameter is declared as a constant. The type of the parameter is the type of the elements of the index type, or the type of the value field of the record type.

A for statement of the form

for v := e1 **to** e2 **do** S

is equivalent to the sequence of statements

v := e1; S; v := succ(v); S; ... ; v := e2; S

and a for statement of the form

for v := e1 **downto** e2 **do** S

is equivalent to the sequence of statements

v := e1; S; v := pred(v); S; ... ; v := e2; S

A record type generator is a record type which has two components with special names: a variable called *value*, and a function called *next*. These names need not be exported. A for statement of the form

for *v* **in** *x.recordTypeGenerator* **do** S **od**

is equivalent to the block

```
begin var crec: x.recordTypeGenerator;
   if crec.next(True) then
      repeat begin const v=crec.value; S end
      until not crec.next(False)
      fi
end
```

The initial and final statements in the declaration of the generator record type can perform any initialization or cleanup which may be appropriate; note that the final statement is executed whenever control finally leaves the for statement, whether normally or via an exit or return statement.

A for statement involving an index type generator, of the form

```
for v in IndexType do S od
```

is equivalent to the block

```
begin var vv:=IndexType.min
while True do
      begin const v=vv; S end
      if vv=IndexType.max then exit fi; vv:=succ(vv)
      od
end
```

If **decreasing** is present, interchange min and max, and replace succ by pred.

Examples:

```
for i in 2 .. 100 do  if a(i)>max then max := a(i) fi end
```

```
for c in Color do Q(c) od
```

```
for relative in Family.members do {members must be a record type local
to Family's type}
      if relative = thisPerson then exit fi
      od
```

9.2.4. With statements

<with statement> ::= **with** <record variable list> **do** <statement>

<record variable list> ::= <record variable>{,<record variable>}

Within the component statement of the with statement, the components (fields) of the record variable specified by the with clause can be denoted by their field identifier only, i.e., without preceding them with the denotation of the entire record variable. The with clause effectively opens the scope containing the field identifiers of the specified record variable, so that the field identifiers may occur as variable identifiers.

Example:

```
with date do
if month = 12 then
    begin month := 1; year := year+1
    end
else month := month+1
```

is equivalent to

```
if date.month = 12 then
    begin date.month := 1; date.year := date.year+1
    end
else date.month := date.month+1
```

No assignments may be made in the qualified statement to any elements of the record variable list. However, assignments are possible to the components of these variables.

9.2.4 Other uses of binding

If a record variable is to be used a number of times in field designators, it is often convenient to bind it to a short identifier.

Example:

```
begin var d == dateTable(i);
if d.month = 12 then d.month := 1; d.year := d.year+1
else d.month := d.month+1 fi
end
```

is equivalent to

```
if dateTable(i).month = 12 then
      dateTable(i).month := 1; dateTable(i).year := dateTable(i).year+1
else dateTable(i).month := dateTable(i).month+1 fi
end
```

and, also equivalent to

```
begin var m == dateTable(i).month; var y == dateTable(i).year;
if m = 12 then m := 1; y := y+1 else m := m+1 fi
end
```

10. Procedure declarations

Procedure declarations serve to define parts of programs and to associate identifiers with them so that they can be activated by procedure statements. A machine-code procedure is exactly like an ordinary procedure, except that its body is a sequence of machine instructions, represented as manifest integer constants according to an implementation-dependent convention.

<procedure declaration> ::= <procedure heading> = <body> |
machine code <procedure heading> =
<code block>

<body> ::= <block> <end identifier> | **forward**

<end identifier> ::= <identifier> | <empty>

<code block> ::= **code** <manifest constant> { ; <manifest constant>}
<end code>

<end code> ::= **end** | **end code** | **end** <identifier>

<block> ::= <label declaration part>
<constant definition part><type definition part>
<variable declaration part>
<procedure and function declaration part>
<statement part>

The *procedure heading* specifies the identifier naming the procedure and the formal parameter identifiers (if any). The parameters are either constant or variable parameters (cf. also 9.1.2). Procedures and functions which are used as parameters to other procedures and functions must have value parameters only.

If the heading is prefixed by **inline**, this is a hint to the compiler that the procedure body should be copied at each call. Such copying tends to result in faster execution, at the expense of a larger object program. The meaning of the program is not changed by the **inline** prefix.

<procedure heading> ::= **procedure** <identifier>; |
procedure <identifier> (<formal parameter section>
{;<formal parameter section>});

<procedure heading> ::= <inline> **procedure** <identifier>
<formal parameter list> <pre assertion>
<post assertion>

<inline> ::= **inline** | <empty>

<formal parameter list> ::= <formal parameter clause> <imports clause>

<formal parameter clause> ::= (<formal parameter section>
{, <formal parameter section>}) | <empty>

<imports clause> ::= **imports** (<import item> {, <import item>}) |
<empty>

<import item> ::= <pervasive> <binding condition> <identifier>

<formal parameter section> ::= <pervasive> <binding condition>
<parameter group>

<parameter group> ::= <identifier>{,<identifier>} : <type definition> <unchecked>

<unchecked> ::= **unchecked** | <empty>

<pre assertion> ::= **pre** <assertion> | <empty>

<post assertion> ::= **post** <assertion> | <empty>

A parameter group or import item without a preceding **const** or **var** implies that its constituents are constants.

A type specification for a formal parameter may have actual parameters which are other formal parameters; thus

procedure *f*(*n*: 0..1000, *a*: **array** (1..*n*) **of** signedInt) ...

is a legal declaration. This procedure might be called as follows:

begin var *aa*: **array** (1..200) **of** signedInt; ... *f*(200, *aa*); ... **end**

Furthermore, in order to reduce the proliferation of parameters which would otherwise be required, we make the following rule: the type of a formal parameter may be a parametrized type with some or all of the actual parameters omitted. The omitted parameters are treated as though they appeared as additional formal parameters, and the appropriate actual parameters are supplied in every call. Thus

procedure *f*(*a*: **array** (1..*n*) **where** (*n*: integer)) ...

is also legal and is equivalent to the previous declaration of *f*, except that all the calls on *f* will be modified appropriately. The previous call would be written

... *f*(*aa*) ...

and would be modified to become

... *f*(*aa*.indexType.max, *aa*) ...

If **unchecked** follows the type definition for a formal parameter group, then an actual parameter of *any* type may be passed. Obviously the language can offer no guarantee of type-safety when this feature is used, and therefore its use should be confined to situations of desperate need.

The label declaration part specifies all labels which mark a statement in the statement part.

<label declaration part> ::= <empty> | **label** <label> {,<label>};

The constant definition part contains all constant synonym definitions local to the procedure.

<constant definition part> ::= <empty> |
const <constant definition> {;<constant definition>};

The type definition part contains all type definitions which are local to the procedure declaration.

<type definition part> ::= <empty> |
type <type definition> {;<type definition>};

The variable declaration part contains all variable declarations local to the procedure declaration.

<variable declaration part> ::= <empty> |
var <variable declaration> {;<variable declaration>};

The procedure and function declaration part contains all procedure and function declarations local to the procedure declaration.

<procedure and function declaration part> ::=
{<procedure or function declaration>;}

<procedure or function declaration> ::=
<procedure declaration> | <function declaration>

The statement part specifies the algorithmic actions to be executed upon an activation of the procedure by a procedure statement.

<statement part> ::= <compound statement>

All identifiers introduced in the formal parameter part, the constant definition part, the type definition part, the variable-, procedure or function declaration parts are local to the procedure declaration which is called the scope of these identifiers. They are not known outside their scope. In the case of local variables, their values are undefined at the beginning of the statement part.

The use of the procedure identifier in a procedure statement within its declaration implies recursive execution of the procedure.

Examples of procedure declarations:

```
type UB = 1 .. 256;  {a type used in the following procedures}
procedure TreeSort(var a: array (1..n) of signedInt where (n: UB)) =
{these two procedures, TreeSort and SiftUp together are a version of
Floyd's TreeSort algorithm in CACM, 7 (1964), p. 701}
    begin
    for i decreasing in 1 .. n div 2 do SiftUp(a, i, n) od;
    for i decreasing in 1 .. n do
        begin const t = a(1);  {swap a(i+1) and a(1) }
        a(1) := a(i+1);
        a(i+1) := t;
        SiftUp(a, 1, i)
        end
        end do
    end TreeSort;
procedure SiftUp(var a: array (1..n) of signedInt where (n: UB, i, j: UB)) =
    begin
    k, l: UB;
    while True do
        begin const k2 = 2*k;
        if k2>j then return fi
        if (k2+1)>j or a(k2+1)<a(k2) then l := k2 else l := k2+1 fi;
        if a(l)<a(k) then return fi;
            begin const t = a(1);  {swap a(i+1) and a(1)}
            a(1) := a(i+1);
            a(i+1) :=t;
            end
        end
        end do
    end SiftUp;
procedure ZeroArray(var a: array (m .. n) of unsignedInt where (m, n:
signedInt)) post {a(m)=0, ... a(n)=0} =
    begin for i in m .. n do a(i) := 0 od
```

end *ZeroArray*

10.1. Standard procedures

Standard procedures are supposed to be predeclared in every implementation of Euclid. Any implementation may feature additional predeclared procedures. Since they are, as all standard quantities, assumed as declared in a scope surrounding the program, no conflict arises from a declaration redefining the same identifier within the program. Standard procedures are pervasive, and hence may not be redeclared. The standard procedures are listed and explained below.

10.1.1. File handling procedures

put(f) appends the value of the buffer variable f↑ to the file f. The effect is defined only if prior to execution the predicate eof(f) is true. eof(f) remains true, and the value of f↑ becomes undefined.

get(f) advances the current file position (read/write head) to the next component, and assigns the value of this component to the buffer variable f↑. If no next component exists, then eof(f) becomes true, and the value of f↑ is not defined. The effect of get(f) is defined only if eof(f) = false prior to its execution. (See 11.1.2.)

reset(f) resets the current file position to its beginning and assigns to the buffer variable f↑ the value of the first element of f. eof(f) becomes false, if f is not empty; otherwise f↑ is not defined, and eof(f) remains true.

rewrite(f) discards the current value of f such that a new file may be generated. eof(f) becomes true.

Concerning the textfile procedures read, write, readln, writeln, and page, see Chapter 12.

10.1.2. Dynamic allocation procedures

new(*p*: ↑*C*) allocates a new variable *v* of type *T* in collection *C* and assigns the pointer to *v* to the pointer variable *p*. New imports *C* as a variable. This procedure works by calling Allocate for the pointer's zone (see 6.3) with the number of StorageUnits required for a variable of type *T*. It gets back a ↑**collection of** StorageBlock, and uses the Storage array in this block for the newly created variable. It is up to the verifier to ensure that this array has at least *n* components if Allocate(*n*) was called, and that the storage allocated does not overlap with that of any other variable (other than one of type StorageUnit). Any initialization specified by the type of *v* is performed. If the object type of *C* is parameterized, and any of the actual parameters are **unknown** or **any**, then specific values for these parameters must be supplied as additional parameters to new, so that the variable being created will have a definite type.

free(*p*: ↑*C*) frees the variable *v* pointed to by *p* and sets *p* to **nil**; there should not be any other pointers equal to *p*. Any finalization specified by the type of *v* is performed. Then the Deallocate procedure for *C*'s zone is called with a pointer to the StorageBlock from which *v* was originally allocated by new.

These two procedures, and the **unchecked** option for formal parameters, are the *only* ways to change the type of a variable. They should be used with due caution.

new(p, t1, ..., tn) can be used to allocate a variable of the variant with tag field values t1, ..., tn.

dispose(p) and dispose(p, t1, ..., tn) can be used to indicate that storage occupied by the variable referenced by the pointer p is no longer needed. (Implementations may use this information to retrieve storage, or they may ignore it.)

10.1.3. Data transfer procedures

Let the variables *a* and *z* be declared by

a: **array** [*m*..*n*] **of** *T*

z: **packed array** [*u*..*v*] **of** *T*

where *n*-*m* >= *v*-*u*. Then the statement pack(*a*, *i*, *z*) means

for *j* **in** *u*..v **do** *z*[*j*] := *a*[*j*-*u*+*i*] **od**

and the statement unpack(*z*, *a*, *i*) means

for *j* **in** *u*..*v* **do** *a*[*j*-*u*+*i*] := *z*[*j*] **od**

where j denotes an auxiliary variable not occurring elsewhere in the program.

11. Function declarations

Function declarations serve to define parts of the program which compute a scalar value or a pointer value. Functions are activated by the evaluation of a function designator (cf. 8.2) which is a constituent of an expression.

⟨function declaration⟩ ::= ⟨function heading⟩ = ⟨body⟩ |
machine code ⟨function heading⟩ = ⟨code block⟩

The function heading specifies the identifier naming the function, the formal parameters of the function, and the type of the function.

⟨function heading⟩ ::= **function** ⟨identifier⟩:⟨result type⟩; |
function ⟨identifier⟩ (⟨formal parameter section⟩
{;⟨formal parameter section⟩}) : ⟨result type⟩;

⟨result type⟩ ::= ⟨type identifier⟩

⟨function heading⟩ ::= ⟨inline⟩ **function** ⟨identifier⟩
⟨formal parameter list⟩
returns ⟨result name⟩ ⟨type definition⟩
⟨pre assertion⟩ ⟨post assertion⟩

⟨result name⟩ ::= ⟨identifier⟩ : | ⟨empty⟩

The type of the function must be a scalar, subrange, or pointer type. Functions may return values of any type except collections. If the result name is supplied, then within the function declaration there may be one or more assignment statements assigning a value to the result name, and the value of the result name when the function returns determines the value of the function. If no result name is supplied, or if it is not assigned to in the body, the result must be supplied in a return statement. A return statement without any value is supplied automatically just before the end of the body. Occurrence of the function identifier in a function designator within its declaration implies recursive execution of the function.

Examples:

```
function Max(a: array (m .. n) of signedInt where (m, n: signedInt))
returns index: signedInt =
        index := m;
        for i in m+1 .. n do
                assert {a(index) = max(a(m), ..., a(i-1))};
                if a(i) > a(index) then index := i fi
                od
        assert {a(index) = max(a(m), ..., a(n))};
        return index
        end Max

function Gcd(m, n: signedInt) returns signedInt =
        begin if n=0 then return m else return Gcd(n, m mod n) fi
        end

function Power(x: signedInt, y: unsignedInt) returns z: signedInt =
        begin var w, i: signedInt;
        w := x;  i := y;  z := 1;
        while i > 0 invariant {z*(w**i) = x**i}  do
```

```
        if odd(i) then z := z*w fi;
        i := i div 2;
        w := w*w
        end do;
    assert {z = x**y};
    return z
    end Power
```

11.1. Standard functions

Standard functions are supposed to be predeclared in every implementation of Euclid. Any implementation may feature additional predeclared functions (cf. also 10.1). Standard functions are pervasive.

The standard functions are listed and explained below:

11.1.1. Arithmetic functions

abs(*x*) computes the absolute value of *x*. The type of *x* must be either real or a subrange of integer, and the type of the result is integer.

sqr(x) computes x**2. The type of x must be either real or integer; the type of the result is the type of x.

sin(x)
cos(x)
exp(x)
ln(x)
sqrt(x)
arctan(x)

the type of x must be either real or integer, and the type of the result is real.

11.1.2. Predicates

odd(*x*) the type of *x* must be a subrange of integer, and the result is True if *x* is odd, and False otherwise.

eof(f) eof(f) indicates whether the file f is in the end-of-file status.

eoln(f) indicates the end of a line in a textfile (see chapter 12).

11.1.3. Transfer functions

trunc(x) the real value x is truncated to its integral part.

round(x) the real argument x is rounded to the nearest integer.

ord(*x*) *x* must be of a scalar type (including Boolean and char), and the result (of type integer) is the ordinal number of the value *x* in the set defined by the type of *x*.

chr(*x*) *x* must be of type integer, and the result (of type char) is the character whose ordinal number is *x* (if it exists).

index(*p*) *p* must be a pointer, and the result, of type unsignedInt, has no properties except that it is guaranteed to be the same if *p* has the same value.

11.1.4. Further standard functions

x.succ — *x* is of any scalar or subrange type, and the result is the successor value of *x* (if it exists).

x.pred — *x* is of any scalar or subrange type, and the result is the predecessor value of *x* (if it exists).

x.max — *x* is of any scalar or subrange type, and the result is the largest value of the type.

x.min — *x* is of any scalar or subrange type, and the result is the smallest value of the type.

12. Input and output

The basis of legible input and output are textfiles (cf. 6.2.4) that are passed as program parameters (cf. 13) to a Pascal program and in its environment represent some input or output device such as a terminal, a card reader, or a line printer. In order to facilitate the handling of textfiles, the four standard procedures read, write, readln, and writeln are introduced in addition to get and put. They can be applied to textfiles only; however, these textfiles must not necessarily represent input/output devices, but can also be local files. The new procedures are used with a non-standard syntax for their parameter lists, allowing, among other things, for a variable number of parameters. Moreover, the parameters must not necessarily be of type char, but may also be of certain other types, in which case the data transfer is accompanied by an implicit data conversion operation. If the first parameter is a file variable, then this is the file to be read or written. Otherwise, the standard files input and output are automatically assumed as default values in the cases of reading and writing respectively. These two files are predeclared as

```
var input, output: text
```

Textfiles represent a special case among file types insofar as texts are substructured into lines by so-called line markers (cf. 6.2.4). If, upon reading a textfile f, the file position is advanced to a line marker that is past the last character of a line, then the value of the buffer variable f↑ becomes a blank, and the standard function eoln(f) (end of line) yields the value true. Advancing the file position once more assigns to f↑ the first character of the next line, and eoln(f) yields false (unless the next line consists of 0 characters). Line markers, not being elements of type char, can only be generated by the procedure writeln.

12.1. The procedure read

The following rules hold for the procedure read; f denotes a textfile and v1...vn denote variables of the types char, integer (or subrange of integer), or real.

1. read(v1, ..., vn) is equivalent to read(input, v1, ..., vn)
2. read(f, v1, ..., vn) is equivalent to read(f, v1); ...; read(f, vn)
3. if v is a variable of type char, then read(f, v) is equivalent to v := f↑; get(f)
4. if v is a variable of type integer (or subrange of integer) or real, then read(f, v) implies the reading from f of a sequence of characters which form a number according to the syntax of Pascal (cf. 4) and the assignment of that number to v. Preceding blanks and line markers are skipped.

12.2. The procedure readln

1. readln(v1, ..., vn) is equivalent to readln(input, v1, ..., vn)
2. readln(f, v1, ..., vn) is equivalent to

   ```
   read(f, v1, ..., vn); readln(f)
   ```
3. readln(f) is equivalent to

   ```
   while not eoln(f) do get(f);
   get(f)
   ```

Readln is used to read and subsequently skip to the beginning of the next line.

12.3. The procedure write

The following rules hold for the procedure write; f denotes a textfile, p1, ..., pn denote so-called write-parameters, e denotes an expression, m and n denote expressions of type integer.

1. write(p1, ..., pn) is equivalent to write(output, p1, ..., pn)
2. write (f, p1, ..., pn) is equivalent to

   ```
   write(f, p1); ...; write(f, pn)
   ```
3. The write-parameters p have the following forms:

   ```
   e:m    e:m:n    e
   ```

e represents the value to be "written" on the file f, and m and n are so-called field width parameters. If the value e, which is either a number, a character, a Boolean value, or a string, requires less than m characters for its representation, then an adequate number of blanks is issued such that exactly m characters are written. If m is omitted, an implementation-defined default

value will be assumed. The form with the width parameter n is applicable only if e is of type real (see rule 6).

4. if e is of type char, then

```
write(f, e:m) is equivalent to
f↑ := ' '; put(f);          (repeated m-1 times)
f↑ := e; put(f)
```

Note: the default value for m is, in this case, 1.

5. If e is of type integer (or a subrange of integer), then the decimal representation of the number e will be written on the file f, preceded by an appropriate number of blanks as specified by m.
6. If e is of type real, a decimal representation of the number e is written on the file f, preceded by an appropriate number of blanks as specified by m. If the parameter n is missing (see rule 3), a floating-point representation consisting of a coefficient and a scale factor will be chosen. Otherwise a fixed-point representation with n digits after the decimal point is obtained.
7. if e is of type Boolean, then the words TRUE or FALSE are written on the file f, preceded by a an appropriate number of blanks as specified by m.
8. if e is an array (packed) of characters, then the string e is written on the file f.

12.4 The procedure writeln

1. writeln(p1, ..., pn) is equivalent to writeln(output, p1, ..., pn)
2. writeln(f, p1, ..., pn) is equivalent to write(f, p1, ..., pn); writeln(f)
3. writeln(f) appends a line marker (cf. 6.2.4) to the file f.

12.5 Additional procedures

page(f) causes skipping to the top of a new page, when the textfile f is printed.

13. Programs

A Euclid program consists of a procedure declaration.

⟨program⟩ ::= ⟨procedure declaration⟩

A Pascal program has the form of a procedure declaration except for its heading.

⟨program⟩ ::= ⟨program heading⟩ ⟨block⟩

⟨program heading⟩ ::= **program** ⟨identifier⟩ (⟨program parameters⟩);

⟨program parameters⟩ ::= ⟨identifier⟩ {,⟨identifier⟩}

The identifier following the symbol **program** is the program name; it has no further significance inside the program. The program parameters denote entities that exist outside the program, and through which the program communicates with its environment. These entities (usually files) are called external, and must be declared in the block which constitutes the program like ordinary local variables.

The two standard files input and output must not be declared (cf. 12), but have to be listed as parameters in the program heading, if they are used. The initializing statements reset(input) and rewrite(output) are automatically generated and must not be specified by the programmer.

Examples:

```
procedure VariousExamples = begin

type HashTable(pervasive size: 1..large)
      imports (pervasive Hash) exports (Search, Delete, Insert, CyclicScan) =

      record
      pervasive type CyclicScan(item: signedInt) exports (Next, value) =
            record
            const start: 1 .. large = Hash(item);
            var value := start;

            function Next(first: Boolean) imports (var value, const start)
            returns Boolean =
                  begin
                  if first then return True fi;
                  if value = size then value := 1 else value := value + 1 fi;
                  return (value not= start)
                  end Next
            end record;

      pervasive type State = (fresh, full, deleted);

      type Entry (var flag: State) exports (key) =
            record
                  case flag of
                        full => (key: signedInt)
            end record;

      var table: array (1 .. size) of Entry;

      function Search(key: signedInt) imports (table) returns Boolean =
            begin
            for i in CyclicScan(key) do
                  case discriminating entry = table(i) of
```

```
                    fresh => return False;
                    full => if entry.key = key then return True fi;
                    end case;
              end do;
        return False
        end Search;

  procedure Delete(key: signedInt) imports (var table) =
        begin
        for i in CyclicScan(key) do
              case discriminating entry = table(i) of
                    full => if entry.key = key then
                    table(i).flag := deleted; return; fi;
                    fresh => return;
                    end case
              end do
        end Delete;


  procedure Insert(key: signedInt) imports (var table, Search) =
        begin
        if Search(key) then return fi;
        for i in CyclicScan(key) do
              case discriminating entry = table(i) of
                    fresh, deleted =>
                          begin
                          table(i).flag := full;
                          case discriminating var new == table(i) of
                                full => new.key := key;
                                end case;
                          return
                          end
              end case
              end do;
        Error('Table is full')  {where is the procedure Error defined?}
        end Insert;

  initially for i in IndexType(table) do table(i) := Entry(fresh) end do
  end HashTable;

type Interval(a, b: signedInt) exports (Next, value) =
      record
      var value := a;
      function Next(first: Boolean) imports (var value, const b) returns
      Boolean =
            begin
            if not first then value := value + 1 fi;
            return (value <= b)
            end Next
      end record;

type StringLength = 0 .. 247;

type Vstring(length: StringLength)
```

```
imports (StringLength) exports (length, var text, Substr, SetSubstr) =

    record
    type StringIndex = 1 .. StringLength.max;
    var text: array (1 .. length) of Character;
    function Substr(start: StringIndex, len: StringLength)
        imports (text, Vstring) returns Vstring =
          begin
          var v: Vstring(len);
          for i in 1 .. len do v.text(i) := text(start+i) end do;
          return v
          end Substr;
    procedure SetSubstr(start: StringIndex, v: Vstring) imports (var text) =
          for i in 1..v.length do text(start+i) := v.text(i) end do;

    initially for i in 1..length do text(i) := "*S

    end Vstring;


function Catenate(to, from: Vstring) imports(Vstring) returns Vstring =
    begin
    var s: Vstring(to.length+from.length);
    s.SetSubstr(1, to); s.SetSubstr(to.length+1, from);
    return s
    end Catenate;

end
```

14. A standard for implementation and program interchange

A primary motivation for the development of Euclid was the need for a powerful and flexible language that could be reasonably efficiently implemented on most computers. Its features were to be defined without reference to any particular machine in order to facilitate the interchange of programs. The following set of proposed restrictions is designed as a guideline for implementors and for programmers who anticipate that their programs be used on different computers. The purpose of these standards is to increase the likelihood that different implementations will be compatible, and that programs are transferable from one installation to another.

1. Identifiers denoting distinct objects must differ over their first 8 characters.

2. Labels consist of at most 4 digits.

3. The implementor may set a limit to the size of a base type over which a set can be defined. (Consequently, a bit pattern representation may reasonably be used for sets.)

4. The first character on each line of printfiles may be interpreted as a printer control character with the following meanings:

   blank single spacing
   '0' double spacing
   '1' print on top of next page

   Representations of Pascal in terms of available character sets should obey the following rules:

5. Word symbols, such as **begin**, **end**, etc., are written as a sequence of letters (without surrounding escape characters). They may not be used as identifiers.

6. Blanks, ends of lines, and comments are considered as separators. An arbitrary number of separators may occur between any two consecutive Euclid symbols with the following restriction: no separators must occur within identifiers, numbers, and word symbols.

7. At least one separator must occur between any pair of consecutive identifiers, numbers, or word symbols.

15. Implementation notes

A later version of this report will include suggestions for implementation techniques which can handle Euclid records and parameterized types efficiently.