USUS NewsLetter

Serving the Pascal, Modula-2, and Portable Programming Community — Vol. 5 No. 2 Mar - Apr 1991

IN THIS ISSUE

From Modula to Oberon	1
The Programming Language Oberon	7
Oberon EBNF	18
Oberon Availability	20
Interupt Routines in JPI Modula-2	22
Board Meeting Minutes (March 13, 1991)	28
Board Meeting Minutes (April 10, 1991)	29
Announcements	30
Treasurer's Report	31
Submission Guidelines	31

Copyright 1991. USUS INC. All Rights Reserved.

The USUS NewsLetter is published ≈6 times per year by USUS, the UCSD Pascal System User's Society, P.O. Box 1148 La Jolla, California 92038. The NewsLetter is a direct benefit of membership in USUS.

Tom Cattrall Editor
William Smith Publisher

From Modula to Oberon

N. Wirth

Abstract

The programming language Oberon is the result of a concentrated effort to increase the power of Modula-2 and simultaneously to reduce its complexity. Several features were eliminated, and a few were added in order to increase the expressive power and flexibility of the language. This paper describes and motivates the changes. The language is defined in a concise report.

Introduction

The programming language Oberon evolved from a project whose goal was the design of a modern, flexible, and efficient operating system for a single-user workstation. A principal guideline was to concentrate on properties that are genuinely essential and - as a consequence - to omit ephemeral issues. It is the best way to keep a system in hand, to make it understandable, explicable, reliable, and efficiently implementable.

Initially, it was planned to express the system in Modula-2 [1] (subsequently called Modula), as that language supports the notion of modular design quite effectively, and because an operating system has to be designed in terms of separately compilable parts with conscientiously chosen interfaces. In fact, an operating system should be no more than a set of basic modules, and the design of an application must be considered as a goal-oriented extension of that basic set: Programming is always extending a given system.

Whereas modern languages, such as Modula, support the notion of extensibility in the procedural realm, the notion is less well established in the domain of data types. In particular, Modula does not allow the definition of new data types as extensions of other, programmer-defined types in an adequate manner. An additional feature was called for, thereby giving rise to an *extension* of Modula.

The concept of the planned operating system also called for a highly dynamic, centralized storage management relying on the technique of garbage collection. Although Modula does not prevent the incorpora-

tion of a garbage collector in principle, its variant record feature constitutes a genuine obstacle. As the new facility for extending types would make the variant record feature superfluous, the removal of this stumbling block was a logical decision. This step, however, gave rise to a *restriction* (subset) of Modula.

It soon became clear that the rule to concentrate on the essential and to eliminate the inessential should not only be applied to the design of the new system, but equally stringently to the language in which the system is formulated. The application of the principle thus led from Modula to a new language. However, the adjective "new" has to be understood in proper context: Oberon evolved from Modula by very few additions and several subtractions. In relying on evolution rather than revolution we remain in the tradition of a long development that led from Algol to Pascal, then to Modula-2, and eventually to Oberon. The common traits of these languages are their procedural rather than functional model and the strict typing of data. Even more fundamental, perhaps, is the idea of abstraction: the language must be defined in terms of mathematical, abstract concepts without reference to any computing mechanism. Only if a language satisfies this criterion, can it be called "higher-level". No syntactic coating whatsoever can earn a language this attribute alone. The definition of a language must be coherent and concise. This can only be achieved by a careful choice of the underlying abstractions and an appropriate structure combining them. The language manual must be reasonably short, avoiding the explanation of individual cases derivable from the general rules. The power of a formalism must not be measured by the length of its description. To the contrary, an overly lengthy definition is a sure symptom of inadequacy. In this respect, not complexity but simplicity must be the goal.

In spite of its brevity, a description must be complete. Completeness is to be achieved within the framework of the chosen abstractions. Limitations imposed by particular implementations do not belong to a language definition proper. Examples of such restrictions are the maximum values of numbers, rounding and truncation errors in arithmetic, and actions taken when a program violates the stated rules. It should not be necessary to supplement a language definition with a voluminous standards document to cover "unforeseen" cases.

But neither should a programming language be a mathematical theory only. It must be a practical tool. This imposes certain limits on the terseness of the formalism. Several features of Oberon are superfluous from a purely theoretical point of view. They are nevertheless retained for practical reasons, either for programmers' convenience or to allow for efficient code generation without the necessity of complex, "optimizing" pattern matching algorithms in compilers. Examples of such features are the presence of several forms of repetitive statements, and of standard procedures such as INC, DEC, and ODD. They complicate neither the language conceptually nor the compiler to any significant degree.

These underlying premises must be kept in mind when comparing Oberon with other languages. Neither the language nor its defining document reach the ideal; but Oberon approximates these goals much better than its predecessors.

A compiler for Oberon has been implemented for the NS32000 processor family and is embedded in the Oberon operating environment [8]. The compiler requires less than 50 KByte of memory, consists of 6 modules with a total of about 4000 lines of source code, and compiles itself in about 15 seconds on a workstation with a 25MHz NS32532 processor.

After extensive experience in programming with Oberon, a revision was defined and implemented. The differences between the two versions are summarised towards the end of the paper. Subsequently, we present a brief introduction to (revised) Oberon assuming familiarity with Modula (or Pascal), concentrating on the added features and listing the eliminated ones. In order to be able to start with a clean slate, the latter are taken first.

Features omitted from Modula

Data types

Variant records are eliminated, because they constitute a genuine difficulty for the implementation of a reliable storage management system based on automatic garbage collection. The functionality of variant records is preserved by the introduction of extensible data types.

Opaque types cater for the concept of abstract data type and information hiding. They are eliminated as such, because again the concept is covered by the new facility of extended record types.

Enumeration types appear to be a simple enough feature to be uncontroversial. However, they defy extensibility over module boundaries. Either a facility to extend given enumeration types has to be introduced, or they have to be dropped. A reason in favour of the latter, radical solution was the observation that in a growing number of programs the indiscriminate use of enumerations (and subranges) had led to a type explosion that contributed not to program clarity but rather to verbosity. In connection with import and export, enumerations give rise to the exceptional rule that the import of a type identifier also causes the (automatic) import of all associated constant identifiers. This exceptional rule defies conceptual simplicity and causes unpleasant problems for the implementor. *Subrange types* were introduced in Pascal (and adopted in Modula) for two reasons: (1) to indicate that a variable accepts a limited range of values of the base type and to allow a compiler to generate appropriate guards for assignments, and (2) to allow a compiler to allocate the minimal storage space needed to store values of the indicated subrange. This appeared desirable in connection with packed records. Very few implementations have taken advantage of this space saving facility, because the additional compiler complexity is very considerable. Reason 1 alone, however, did not appear to provide sufficient justification to retain the subrange facility in Oberon.

With the absence of enumeration and subrange types, the general possibility of defining *set types* based on given element types appeared as redundant. Instead, a single, basic type SET is introduced, whose values are sets of integers from 0 to an implementation-defined maximum.

The basic type *CARDINAL* had been introduced in Modula in order to allow address arithmetic with values from 0 to 2 16 on 16-bit computers. With the prevalence of 32-bit addresses in modern processors, the need for unsigned arithmetic has practically vanished, and therefore the type CARDINAL has been eliminated. With it, the bothersome incompatibilities of operands of types CARDINAL and INTEGER have disappeared.

Pointer types are restricted to be bound to a record type or to an array type.

The notion of a definable index type of arrays has also been abandoned: All indices are by default integers. Furthermore, the lower bound is fixed to 0; array declarations specify a number of elements (length) rather than a pair of bounds. This break with a long standing tradition since Algol 60 clearly demonstrates the principle of eliminating the inessential. The specification of an arbitrary lower bound hardly provides any additional expressive power. It represents a rather limited kind of mapping of indices which introduces a hidden computational effort that is incommensurate with the supposed gain in convenience. This effort is particularly heavy in connection with bound checking and with dynamic arrays.

Modules and import/export rules

Experience with Modula over the last eight years has shown that *local modules* were rarely used. Considering the additional complexity of the compiler required to handle them, and the additional complications in the visiblity rules of the language definition, the elimination of local modules appears justified.

The *qualification* of an imported object's identifier x by the exporting module's name M, viz. M.x, can be circumvented in Modula by the use of the import clause FROM M IMPORT x. This facility has also been discarded. Experience in programming systems involving many modules has taught that the explicit qualification of each occurrence of x is actually preferable. A simplification of the compiler is a welcome side-effect.

The dual role of the main module in Modula is conceptually confusing. It constitutes a *module* in the sense of a package of data and procedures enclosed by a scope of visibility, and at the same time it constitutes a single *procedure* called main program. A module is composed of two textual pieces, called the definition part and the implementation part. The former is missing in the case of a main program module.

By contrast, a module in Oberon is in itself complete and constitutes a unit of compilation. Definition and implementation parts are merged; names to be visible in client modules, i.e. exported identifiers, are marked, and they typically precede the declarations of objects not exported. A compilation generates in general a changed object file and a new symbol file. The latter contains information about exported objects for use in the compilation of client modules. The generation of a new symbol file must, however, be specifically enabled by a compiler option, because it will invalidate previous compilations of clients.

The notion of a main program has been abandoned. Instead, the set of modules linked through imports typically contains (parameterless) procedures. They are to be considered as individually activatable, and they are called *commands.* Such an activation has the form M.P, where P denotes the command and M the module containing it. The effect of a command is considered - not like that of a main program as accepting input and transforming it to output - as a change of state represented by global data.

Statements

The *with statement* of Modula has been discarded. Like in the case of imported identifiers, the explicit qualification of field identifiers is to be preferred. Another form of with statement is introduced; it has a different function and is called a regional guard (see below).

The elimination of the for statement constitutes a break with another long standing tradition. The baroque mechanism of Algol 60's for statement had been trimmed significantly in Pascal (and Modula). Its marginal value in practice has led to its absence from Oberon.

Low-level facilities

Modula makes access to machine-specific facilities possible through low-level constructs, such as the data types ADDRESS and WORD, absolute addressing of variables, and type casting functions. Most of them are packaged in a module called SYSTEM. These features were supposed to be rarely used and easily visible through the presence of the identifier SYSTEM in a module's import list. Experience has revealed, however, that a significant number of programmers import this module quite indiscriminately. A particularly seductive trap are Modula's type transfer functions.

It appears preferable to drop the pretense of portability of programs that import a "standard", yet system-specific module. *Type transfer functions* denoted by type identifiers are therefore eliminated, and the module SYSTEM is restricted to providing a few machine-specific functions that typically are compiled into inline code. The types *ADDRESS* and *WORD* are replaced by the type BYTE, for which type compatibility rules are relaxed. Individual implementations are free to provide additional facilities in their module SYSTEM. The use of SYSTEM declares a program to be patently implementation-specific and thereby non-portable.

Concurrency

The system Oberon does not require any language facilities for expressing concurrent processes. The pertinent rudimentary features of Modula, in particular the coroutine, were therefore not retained. This exclusion is merely a reflection of our actual needs within the concrete project, but not on the general relevance of concurrency in programming.

Features introduced in Oberon

In contrast to the number of eliminated features, there are only a few new ones. The important new concepts are type exten-

sion and type inclusion. Furthermore, open arrays may have several dimensions (indices), whereas in Modula they were confined to a single dimension.

Type extension

The most important addition is the facility of extended record types. It permits the construction of new types on the basis of existing types, and establishes a certain degree of compatibility between the new and old types. Assuming a given type

```
T = RECORD x, y: INTEGER END
```

extensions may be defined which contain certain fields in addition to the existing ones. For example

```
T0 = RECORD (T)  z: REAL END
T1 = RECORD (T)  w: LONGREAL END
```

define types with fields x, y, z and x, y, w respectively. We define a type declared by

```
T' = RECORD (T) <field definitions> END
```

to be a *(direct) extension* of T, and conversely T to be the *(direct) base type* of T'. Extended types may be extended again, giving rise to the following definitions:

A type T' is an *extension* of T, if T' = T or T' is a direct extension of an extension of T. Conversely, T is a *base type* of T', if T = T' or T is the direct base type of a base type of T'. We denote this relationship by T' <= T.

The rule of assignment compatibility states that values of an extended type are assignable to variables of their base types. For example, a record of type T0 can be assigned to a variable of the base type T. This assignment involves the fields x and y only, and in fact constitutes a *projection* of the value onto the space spanned by the base type.

It is important to allow modules which import a base type to be able to declare extended types. In fact, this is probably the normal usage.

This concept of extensible data type gains importance when extended to pointers. It is appropriate to say that a pointer type P' bound to T' extends a pointer type P, if P is bound to a base type T of T', and to extend the assignment rule to cover this case. It is now possible to form data structures whose nodes are of different types, i.e. inhomogeneous data structures. The inhomogeneity is automatically (and most sensibly) bounded by the fact that the nodes are linked by pointers of a common base type.

Typically, the pointer fields establishing the structure are contained in the base type T, and the procedures manipulating the structure are defined in the same (base) module as T. Individual extensions (variants) are defined in client modules together with procedures operating on nodes of the extended type. This scheme is in full accordance with the notion of system extensibility: new modules defining new extensions may be added to a system without requiring a change of the base modules, not even their recompilation.

As access to an individual node via a pointer bound to a base type provides a projected view of the node data only, a facility to widen the view is necessary. It depends on the ability to determine the actual type of the referenced node. This is achieved by a *type test*, a Boolean expression of the form

```
t IS T' (or  p IS P')
```

If the test is affirmative, an assignment t' := t (t' of type T') or p' := p (p' of type P') should be possible. The static view of types, however, prohibits this. Note that both assignments violate the rule of assignment compatibility. The desired assignment is made possible by providing a *type guard* of the form

```
t' := t(T') (p' := p(P'))
```

and by the same token access to the field z of a T0 (see previous examples) is made possible by a type guard in the designator t(T0).z. Here the guard asserts that t is (currently) of type T0. In analogy to array bound checks and case selectors, a failing guard leads to program abortion.

Whereas a guard of the form t(T) asserts that t is of type T for the designator (starting with) t only, a *regional type guard* maintains the assertion over an entire sequence of statements. It has the form

```
WITH t: T DO StatementSequence END
```

and specifies that t is to be regarded as of type T within the entire statement sequence. Typically, T is an extension of the declared type of t. Note that assignments to t within the region therefore require the assigned value to be (an extension) of type T. The regional guard serves to reduce the number of guard evaluations.

As an example of the use of type tests and guards, consider the following types Node and Object defined in a module M:

```
TYPE Node = POINTER TO Object;
       Object = RECORD key, x, y: INTEGER;
                  left, right: Node
       END
```

Elements in a tree structure anchored in a variable called root (of type Node) are searched by the procedure *element* defined in M.

```
PROCEDURE element(k: INTEGER): Node;
  VAR p: Node;
BEGIN p := root;
  WHILE (p # NIL) & (p.key # k) DO
    IF p.key < k THEN
      p := p.left ELSE p := p.right END
  END ;
  RETURN p
END element
```

Let extensions of the type Object be defined (together with their pointer types) in a module M1 which is a client of M:

```
TYPE
  Rectangle = POINTER TO RectObject;
  RectObject = RECORD (Object) w, h: REAL END ;
```

```
Circle = POINTER TO CircleObject;
CircleObject = RECORD (Object)
                    rad: REAL;
                    shaded: BOOLEAN
               END
```

After the search of an element, the type test is used to discriminate between the different extensions, and the type guard to access extension fields. For example:

```
p := M.element(K);
IF p # NIL THEN
   IF p IS Rectangle THEN ... p(Rectangle).w ...
   ELSIF (p IS Circle) & ~p(Circle).shaded THEN
       ... p(Circle).rad ...
   ELSIF ...
```

The extensibility of a system rests upon the premise that new modules defining new extensions may be added without requiring adaptations nor even recompilation of the existing parts, although components of the new types are included in already existing data structures.

The type extension facility not only replaces Modula's variant records, but represents a type-safe alternative. Equally important is its effect of relating types in a type hierarchy. We compare, for example, the Modula types

```
T0' = RECORD t: T;  z: REAL END ;
T1' = RECORD t: T;  w: LONGREAL END
```

which refer to the definition of T given above, with the extended Oberon types T0 and T1 defined above. First, the Oberon types refrain from introducing a new naming scope. Given a variable r0 of type T0, we write r0.x instead of r0.t.x as in Modula. Second, the types T, T0', and T1' are distinct and unrelated. In contrast, T0 and T1 are related to T as extensions. This becomes manifest through the type test, which asserts that variable r0 is not only of type T0, but also of base type T.

The declaration of extended record types, the type test, and the type guard are the only additional features introduced in this context. A more extensive discussion is provided in [2]. The concept is very similar to the class notion of Simula 67 [3], Smalltalk [4], Object Pascal [5], C++ [6], and others, where the properties of the base class are said to be *inherited* by the derived classes. The class facility stipulates that all procedures applicable to objects of the class be defined together with the data definition. This dogma stems from the notion of abstract data type, but it is a serious obstacle in the development of large systems, where the possibility to add further procedures defined in additional modules is highly desirable. It is awkward to be obliged to redefine a class solely because a method (procedure) has been added or changed, particularly when this change requires a recompilation of the class definition and of all its client modules.

We emphasise that the type extension facility - although gaining its major role in connection with pointers to build heterogeneous, dynamic data structures as shown in the example above - also applies to statically declared objects used as variable parameters. Such objects are allocated in a workspace organized as a stack of procedure activation records, and therefore take advantage of an extremely efficient allocation and deallocation scheme.

In Oberon, procedure *types* rather than procedures (methods) are connected with objects in the program text. The binding of actual methods (specific procedures) to objects (instances) is delayed until the program is executed. The association of a procedure type with a data type occurs through the declaration of a record field. This field is given a procedure type. The association of a method - to use Smalltalk terminology - with an object occurs through the assignment of a specific procedure as value to the field, and not through a static declaration in the extended type's definition which then "overrides" the declaration given in the base type. Such a procedure is called a *handler.* Using type tests, the handler is capable of discriminating among different extensions of the record's (object's) base type. In Smalltalk, the compatibility rules between a class and its subclasses are confined to pointers, thereby intertwining the concepts of access method and data type in an undesirable way. In Oberon, the relationship between a type and its extensions is based on the established mathematical concept of projection.

In Modula, it is possible to declare a pointer type within an implementation module, and to export it as an opaque type by listing the same identifier in the corresponding definition module. The net effect is that the type is exported while all its properties remain hidden (invisible to clients). In Oberon, this facility is generalized in the sense that the selection of the record fields to be exported is arbitrary and includes the cases all and none. The collection of exported fields defines a partial view - a *public projection* - to clients.

In client modules as well as in the module itself, it is possible to define extensions of the base type (e.g. TextViewers or GraphViewers). Of importance is also the fact that non-exported components (fields) may have types that are not exported either. Hence, it is possible to hide certain data types effectively, although components of (opaquely) exported types refer to them.

Type inclusion

Modern processors feature arithmetic operations on several number formats. It is desirable to have all these formats reflected in the language as basic types. Oberon features five of them:

```
LONGINT, INTEGER, SHORTINT (integer types)
LONGREAL, REAL (real types)
```

With the proliferation of basic types, a relaxation of compatibility rules among them becomes almost mandatory. (Note that in Modula the numeric types INTEGER, CARDINAL, and REAL are incompatible). To this end, the notion of *type inclusion* is introduced: a type T includes a type T', if the values of type T' are also values of type T. Oberon postulates the following hierarchy:

```
LONGREAL >= REAL >= LONGINT >=
INTEGER >= SHORTINT
```

The assignment rule is relaxed accordingly: A value of type T' can be assigned to a variable of type T, if T' is included in T

(or if T' extends T), i.e. if T >= T' or T' -> T. In this respect, we return to (and extend) the flexibility of Algol 60. For example, given variables

```
i: INTEGER; k: LONGINT; x: REAL
```

the assignments

```
k := i; x := k; x := 1; k := k+i; x := x*10 + i
```

conform to the rules, whereas the statements i := k; k := x are not acceptable. x := k may involve truncation.

The presence of several numeric types is evidently a concession to implementations which can allocate different amounts of storage to variables of the different types, and which thereby offer an opportunity for storage economization. This practical aspect should - with due respect for mathematical abstraction - not be ignored. The notion of type inclusion minimises the consequences for the programmer and requires only few implicit instructions for changing the data representation, such as sign extensions and integer to floating-point conversions.

Differences between Oberon and Revised Oberon

A revision of Oberon was defined after extensive experience in the use and implementation of the language. Again, it is characterized by the desire to simplify and integrate. The differences between the original version [7] and the revised version [9] are the following:

1. Definition and implementation parts of a module are merged. It appeared as desirable to have a module's specification contained in a single document, both from the view of the programmer and the compiler. A specification of its interface to clients (the definition part) can be derived automatically. Objects previously declared in the definition part (and repeated in the implementation part), are specially marked for export. The need for a structural comparison of two texts by the compiler thereby vanishes.

2. The syntax of lists of parameter types in the declaration of a procedure type is the same as that for regular procedure headings. This implies that dummy identifiers are introduced; they may be useful as comments.

3. The rule that type declarations must follow constant declarations, and that variable declarations must follow type declarations is relaxed.

4. The apostrophe is eliminated as a string delimiter.

5. The relaxed parameter compatibility rule for the formal type ARRAY OF BYTE is applicable for variable parameters only.

Summary

The language Oberon has evolved from Modula-2 and incorporates the experiences of many years of programming in Modula. A significant number of features have been eliminated. They appear to have contributed more to language and compiler complexity than to genuine power and flexibility of expression. A small number of features have been added, the most significant one being the concept of type extension.

The evolution of a new language that is smaller, yet more powerful than its ancestor is contrary to common practices and trends, but has inestimable advantages. Apart from simpler compilers, it results in a concise defining document [9], an indispensible prerequisite for any tool that must serve in the construction of sophisticated and reliable systems.

Acknowlegement

It is impossible to explicitly acknowledge all contributions of ideas that ultimately simmered down to what is now Oberon. Most came from the use or study of existing languages, such as Modula-2, Ada, Smalltalk, and Cedar, which often taught us how *not* to do it. Of particular value was the contribution of Oberon's first user, J. Gutknecht. The author is grateful for his insistence on the elimination of dead wood and on basing the remaining features on a sound mathematical foundation. And last, thanks go to the anonymous referee who very carefully read the manuscript and contributed many valuable suggestions for improvement.

References

1. N. Wirth. *Programming in Modula-2.* Springer-Verlag, 1982.

2. N. Wirth. Type Extensions. *ACM Trans. on Prog. Languages and Systems, 10,* 2 (April 1988) 204-214.

3. G. Birtwistle, et al. *Simula Begin.* Auerbach, 1973.

4. A. Goldberg, D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

5. L. Tesler. Object Pascal Report. *Structured Language World, 9,* 3 (1985), 10-14.

6. B. Stroustrup. *The Programming Language C++.* Addison-Wesley, 1986.

7. N. Wirth. The programming language Oberon. *Software - Practice and Experience, 18,* 7 (July 1988), 671-690.

8. J. Gutknecht and N. Wirth. The Oberon System. *Software - Practice and Experience, 19,* (1989)

9. N. Wirth. The programming language Oberon (Revised Report). (companion paper)

File: ModToOberon2.Doc / NW 1.10.90

The Programming Language Oberon

(Revision 1. 10. 90)

N.Wirth

Make it as simple as possible, but not simpler.
A. Einstein

1. Introduction

Oberon is a general-purpose programming language that evolved from Modula-2. Its principal new feature is the concept of *type extension.* It permits the construction of new data types on the basis of existing ones and to relate them.

This report is not intended as a programmer's tutorial. It is intentionally kept concise. Its function is to serve as a reference for programmers, implementors, and manual writers. What remains unsaid is mostly left so intentionally, either because it is derivable from stated rules of the language, or because it would require to commit the definition when a general commitment appears as unwise.

2. Syntax

A language is an infinite set of sentences, namely the sentences well formed according to its syntax. In Oberon, these sentences are called compilation units. Each unit is a finite sequence of symbols from a finite vocabulary. The vocabulary of Oberon consists of identifiers, numbers, strings, operators, delimiters, and comments. They are called lexical symbols and are composed of sequences of characters. (Note the distinction between symbols and characters.)

To describe the syntax, an extended Backus-Naur Formalism called EBNF is used. Brackets [and] denote optionality of the enclosed sentential form, and braces { and } denote its repetition (possibly 0 times). Syntactic entities (non-terminal symbols) are denoted by English words expressing their intuitive meaning. Symbols of the language vocabulary (terminal symbols) are denoted by strings enclosed in quote marks or words written in capital letters, so-called reserved words. Syntactic rules (productions) are marked by a $ sign at the left margin of the line.

3. Vocabulary and representation

The representation of symbols in terms of characters is defined using the ASCII set. Symbols are identifiers, numbers, strings, operators, delimiters, and comments. The following lexical rules must be observed. Blanks and line breaks must not occur within symbols (except in comments, and blanks in strings). They are ignored unless they are essential to separate two consecutive symbols. Capital and lower-case letters are considered as being distinct.

1. *Identifiers* are sequences of letters and digits. The first character must be a letter.

```
$ ident = letter {letter | digit}.
```

Examples:

```
x  scan  Oberon  GetSymbol  firstLetter
```

2. *Numbers* are (unsigned) integers or real numbers. Integers are sequences of digits and may be followed by a suffix letter. The type is the minimal type to which the number belongs (see 6.1.). If no suffix is specified, the representation is decimal. The suffix H indicates hexadecimal representation.

A real number always contains a decimal point. Optionally it may also contain a decimal scale factor. The letter E (or D) is pronounced as "times ten to the power of". A real number is of type REAL, unless it has a scale factor containing the letter D; in this case it is of type LONGREAL.

```
$    number = integer | real.
$    integer = digit {digit} | digit {hexDigit} "H" .
```

```
$ real = digit {digit} "." {digit} [ScaleFactor].
$ ScaleFactor = ("E" | "D") ["+" | "-"] digit {digit}.
$ hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
$ digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

Examples:

```
1987
100H            = 256
12.3
4.567E8         = 456700000
0.57712566D-6   = 0.00000057712566
```

3. *Character constants* are either denoted by a single character enclosed in quote marks or by the ordinal number of the character in hexadecimal notation followed by the letter X.

```
$ CharConstant = """ character """ | digit {hexDigit} "X".
```

4. *Strings* are sequences of characters enclosed in quote marks ("). A string cannot contain a quote mark. The number of characters in a string is called the *length* of the string. Strings can be assigned to and compared with arrays of characters (see 9.1 and 8.2.4).

```
$ string = """ {character} """ .
```

Examples:

```
"OBERON"   "Don't worry!"
```

5. *Operators and delimiters* are the special characters, character pairs, or *reserved words* listed below. These reserved words consist exclusively of capital letters and cannot be used in the role of identifiers.

```
+    :=    ARRAY    IS          TO
-    ^     BEGIN    LOOP        TYPE
*    =     CASE     MOD         UNTIL
/    #     CONST    MODULE      VAR
~    <     DIV      NIL         WHILE
&    >     DO       OF          WITH
.    <=    ELSE     OR
,    >=    ELSIF    POINTER
;    ..    END      PROCEDURE
|    :     EXIT     RECORD
(    )     IF       REPEAT
[    ]     IMPORT   RETURN
{    }     IN       THEN
```

6. *Comments* may be inserted between any two symbols in a program. They are arbitrary character sequences opened by the bracket (* and closed by *). Comments do not affect the meaning of a program.

4. Declarations and scope rules

Every identifier occurring in a program must be introduced by a declaration, unless it is a predefined identifier. Declarations also serve to specify certain permanent properties of an object, such as whether it is a constant, a type, a variable, or a procedure.

The identifier is then used to refer to the associated object. This is possible in those parts of a program only which are within the *scope* of the declaration. No identifier may denote more than one object within a given scope. The scope extends textually from the point of the declaration to the end of the block (procedure or module) to which the declaration belongs and hence to which the object is *local.* The scope rule has the following amendments:

1. If a type T is defined as POINTER TO T1 (see 6.4), the identifier T1 can be declared textually following the declaration of T, but it must lie within the same scope.

2. Field identifiers of a record declaration (see 6.3) are valid in field designators only.

In its declaration, an identifier in the global scope may be followed by an export mark (*) to indicate that it be exported from its declaring module. In this case, the identifier may be used in other modules, if they import the declaring module. The identifier is then prefixed by the identifier designating its module (see Ch. 11). The prefix and the identifier are separated by a period and together are called a *qualified identifier.*

```
$ qualident = [ident "."] ident.
$ identdef = ident ["*"].
```

The following identifiers are predefined; their meaning is defined in the indicated sections:

ABS	(10.2)	LEN	(10.2)
ASH	(10.2)	LONG	(10.2)
BOOLEAN	(6.1)	LONGINT	(6.1)
BYTE	(6.1)	LONGREAL	(6.1)
CAP	(10.2)	MAX	(10.2)
CHAR	(6.1)	MIN	(10.2)
CHR	(10.2)	NEW	(6.4)
DEC	(10.2)	ODD	(10.2)
ENTIER	(10.2)	ORD	(10.2)

EXCL	(0.2)	SET	(6.1)
HALT	(10.2)	SHORT	(10.2)
INC	(10.2)	SHORTINT	(6.1)
INCL	(10.2)	SIZE	(10.2)
INTEGER	(6.1)	TRUE	(6.1)

5. Constant declarations

A constant declaration associates an identifier with a constant value.

```
$ ConstantDeclaration = identdef "=" ConstExpression.
$ ConstExpression = expression.
```

A constant expression can be evaluated by a mere textual scan without actually executing the program. Its operands are constants (see Ch. 8). Examples of constant declarations are

```
N = 100
limit = 2*N -1
all = {0 .. WordSize-1}
```

6. Type declarations

A data type determines the set of values which variables of that type may assume, and the operators that are applicable. A type declaration is used to associate an identifier with the type. Such association may be with unstructured (basic) types, or it may be with structured types, in which case it defines the structure of variables of this type and, by implication, the operators that are applicable to the components. There are two different structures, namely arrays and records, with different component selectors.

```
$ TypeDeclaration = identdef "=" type.
$ type = qualident | ArrayType | RecordType |
$           PointerType | ProcedureType.
```

Examples:

```
Table = ARRAY N OF REAL

Tree = POINTER TO Node

Node = RECORD key: INTEGER;
           left, right: Tree
        END

CenterNode = RECORD (Node)
                 name: ARRAY 32 OF CHAR;
                 subnode: Tree
               END

Function* = PROCEDURE (x: INTEGER): INTEGER
```

6.1. Basic types

The following basic types are denoted by predeclared identifiers. The associated operators are defined in 8.2, and the predeclared function procedures in 10.2. The values of a given basic type are the following:

1. BOOLEAN	the truth values TRUE and FALSE.
2. CHAR	the characters of the extended ASCII set (0X ... 0FFX).
3. SHORTINT	the integers between -128 and 127.
4. INTEGER	the integers between MIN(INTEGER) and MAX(INTEGER).
5. LONGINT	the integers between MIN(LONGINT) and MAX(LONGINT).
6. REAL	real numbers between MIN(REAL) and MAX(REAL).
7. LONGREAL	real numbers between MIN(LONGREAL) and MAX(LONGREAL).
8. SET	the sets of integers between 0 and MAX(SET).

Types 3 to 5 are *integer* types, 6 and 7 are *real* types, and together they are called *numeric* types. They form a hierarchy; the larger type *includes* (the values of) the smaller type:

```
LONGREAL >= REAL >=
LONGINT >= INTEGER >= SHORTINT
```

6.2. Array types

An array is a structure consisting of a fixed number of elements which are all of the same type, called the *element type*. The number of elements of an array is called its *length*. The elements of the array are designated by indices, which are integers between 0 and the length minus 1.

```
$ ArrayType = ARRAY length {"," length} OF type.
$ length = ConstExpression.
```

A declaration of the form

```
ARRAY N0, N1, ... , Nk OF T
```

is understood as an abbreviation of the declaration

```
ARRAY N0 OF
 ARRAY N1 OF ...
  ARRAY Nk OF T
```

Examples of array types:

```
ARRAY N OF INTEGER
ARRAY 10, 20 OF REAL
```

6.3. Record types

A record type is a structure consisting of a fixed number of elements of possibly different types. The record type declaration specifies for each element, called *field,* its type and an identifier which denotes the field. The scope of these field identifiers is the record definition itself, but they are also visible within field designators (see 8.1) referring to elements of record variables.

```
$ RecordType = RECORD ["(" BaseType ")"]
               FieldListSequence END.
$ BaseType = qualident.
$ FieldListSequence = FieldList {";" FieldList}.
$ FieldList = [IdentList ":" type].
$ IdentList = identdef {"," identdef}.
```

If a record type is exported, field identifiers that are to be visible outside the declaring module must be marked. They are called *public fields* ; unmarked fields are called *private fields.* Record types are extensible, i.e. a record type can be defined as an extension of another record type. In the examples above, CenterNode *(directly) extends* Node, which is the *(direct) base type* of CenterNode. More specifically, CenterNode extends Node with the fields *name* and *subnode.*

Definition: A type T0 *extends* a type T, if it equals T, or if it directly extends an extension of T. Conversely, a type T is a *base type* of T0, if it equals T0, or if it is the direct base type of a base type of T0.

Examples of record types:

```
RECORD day, month, year: INTEGER
END

RECORD
  name, firstname: ARRAY 32 OF CHAR;
  age: INTEGER;
  salary: REAL
END
```

6.4. Pointer types

Variables of a pointer type P assume as values pointers to variables of some type T. The pointer type P is said to be *bound* to T, and T is the *pointer base type* of P. T must be a record or array type. Pointer types inherit the extension relation of their base types. If a type T0 is an extension of T and P0 is a pointer type bound to T0, then P0 is also an extension of P.

```
$ PointerType = POINTER TO type.
```

If p is a variable of type P = POINTER TO T, then a call of the predefined procedure NEW(p) has the following effect (see 10.2): A variable of type T is allocated in free storage, and a pointer to it is assigned to p. This pointer p is of type P; the *referenced* variable p^ is of type T. Failure of allocation results in p obtaining the value NIL. Any pointer variable may be assigned the value NIL, which points to no variable at all.

6.5. Procedure types

Variables of a procedure type T have a procedure (or NIL) as value. If a procedure P is assigned to a procedure variable of type T, the (types of the) formal parameters of P must be the same as those indicated in the formal parameters of T. The same holds for the result type in the case of a function procedure (see 10.1). P must not be declared local to another procedure, and neither can it be a predefined procedure.

```
$ ProcedureType = PROCEDURE [FormalParameters].
```

7. Variable declarations

Variable declarations serve to introduce variables and associate them with identifiers that must be unique within the given scope. They also serve to associate fixed data types with the variables.

```
$ VariableDeclaration = IdentList ":" type.
```

Variables whose identifiers appear in the same list are all of the same type. Examples of variable declarations (refer to examples in Ch. 6):

```
i, j, k: INTEGER
x, y: REAL
p, q: BOOLEAN
s:  SET
```

```
f:  Function
a:  ARRAY 100 OF REAL
w:  ARRAY 16 OF
        RECORD ch: CHAR;
          count: INTEGER
        END
t: Tree
```

8. Expressions

Expressions are constructs denoting rules of computation whereby constants and current values of variables are combined to derive other values by the application of operators and function procedures. Expressions consist of operands and operators. Parentheses may be used to express specific associations of operators and operands.

8.1. Operands

With the exception of sets and literal constants, i.e. numbers and character strings, operands are denoted by *designators*. A designator consists of an identifier referring to the constant, variable, or procedure to be designated. This identifier may possibly be qualified by module identifiers (see Ch. 4 and 11), and it may be followed by selectors, if the designated object is an element of a structure.

If A designates an array, then A[E] denotes that element of A whose index is the current value of the expression E. The type of E must be an integer type. A designator of the form A[E1, E2, ... , En] stands for A[E1][E2] ... [En]. If p designates a pointer variable, p^ denotes the variable which is referenced by p. If r designates a record, then r.f denotes the field f of r. If p designates a pointer, p.f denotes the field f of the record p^, i.e. the dot implies dereferencing and p.f stands for p^.f, and p[E] denotes the element of p^ with index E.

The *typeguard* v(T0) asserts that v is of type T0, i.e. it aborts program execution, if it is not of type T0. The guard is applicable, if

1. T0 is an extension of the declared type T of v, and if

2. v is a variable parameter of record type or v is a pointer.

```
$  designator  =  qualident {"." ident |
$        "[" ExpList "]" | "(" qualident ")" | "^" }.
$  ExpList  =  expression {"," expression}.
```

If the designated object is a variable, then the designator refers to the variable's current value. If the object is a procedure, a designator without parameter list refers to that procedure. If it is followed by a (possibly empty) parameter list, the designator implies an activation of the procedure and stands for the value resulting from its execution. The (types of the) actual parameters must correspond to the formal parameters as specified in the procedure's declaration (see Ch. 10).

Examples of designators (see examples in Ch. 7):

i	(INTEGER)
a[i]	(REAL)
w[3].ch	(CHAR)
t.key	(INTEGER)
t.left.right	(Tree)
t(CenterNode).subnode	(Tree)

8.2. Operators

The syntax of expressions distinguishes between four classes of operators with different precedences (binding strengths). The operator ~ has the highest precedence, followed by multiplication operators, addition operators, and relations. Operators of the same precedence associate from left to right. For example, x-y-z stands for (x-y)-z.

```
$ expression = SimpleExpression [relation SimpleExpression].
$ relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
$ SimpleExpression = ["+"|"-"] term {AddOperator term}.
$ AddOperator = "+" | "-" | OR .
$ term = factor {MulOperator factor}.
$ MulOperator = "*" | "/" | DIV | MOD | "&" .
$ factor = number | CharConstant | string | NIL | set |
$ designator [ActualParameters] | "(" expression ")" | "~" factor.
$ set = "{" [element {"," element}] "}".
$ element = expression [".." expression].
$ ActualParameters = "(" [ExpList] ")" .
```

The available operators are listed in the following tables. In some instances, several different operations are designated by the same operator symbol. In these cases, the actual operation is identified by the type of the operands.

8.2.1. Logical operators

symbol	result
OR	logical disjunction
&	logical conjunction
~	negation

These operators apply to BOOLEAN operands and yield

a BOOLEAN result.

p OR q	stands for "if p then TRUE, else q"
p & q	stands for "if p then q, else FALSE"
~ p	stands for "not p"

8.2.2. Arithmetic operators

symbol	result
+	sum
-	difference
*	product
/	quotient
DIV	integer quotient
MOD	modulus

The operators +, -, *, and / apply to operands of numeric types. The type of the result is that operand's type which includes the other operand's type, except for division (/), where the result is the real type which includes both operand types. When used as operators with a single operand, - denotes sign inversion and + denotes the identity operation.

The operators DIV and MOD apply to integer operands only. They are related by the following formulas defined for any dividend x and positive divisors y:

```
x = (x DIV y) * y + (x MOD y)
0 <= (x MOD y) < y
```

8.2.3. Set operators

symbol	result
+	union
-	difference
*	intersection
/	symmetric set difference

The monadic minus sign denotes the complement of x, i.e. -x denotes the set of integers between 0 and MAX(SET) which are not elements of x.

```
x - y = x * (-y)
x / y = (x-y) + (y-x)
```

8.2.4. Relations

symbol	relation
=	equal
#	unequal
<	less
<=	less or equal
>	greater
>=	greater or equal
IN	set membership
IS	type test

Relations are Boolean. The ordering relations <, <=, >, and >= apply to the numeric types, CHAR, and character arrays (strings). The relations = and # also apply to the type BOOLEAN and to set, pointer, and procedure types. *x IN s* stands for "x is an element of s". x must be of an integer type, and s of type SET. *v IS T* stands for "v is of type T" and is called a *type test.* It is applicable, if

1. T is an extension of the declared type T0 of v, and if

2. v is a variable parameter of record type or v is a pointer.

Assuming, for instance, that T is an extension of T0 and that v is a designator declared of type T0, then the test "v IS T" determines whether the actually designated variable is (not only a T0, but also) a T. The value of NIL IS T is undefined. Examples of expressions (refer to examples in Ch. 7):

1987	(INTEGER)
i DIV 3	(INTEGER)
~p OR q	(BOOLEAN)
(i+j) * (i-j)	(INTEGER)
s - {8, 9, 13}	(SET)
i + x	(REAL)
a[i+j] * a[i-j]	(REAL)
(0<=i) & (i<100)	(BOOLEAN)
t.key = 0	(BOOLEAN)
k IN {i .. j-1}	(BOOLEAN)
t IS CenterNode	(BOOLEAN)

9. Statements

Statements denote actions. There are elementary and structured statements. Elementary statements are not composed of any parts that are themselves statements. They are the assignment, the procedure call, and the return and exit statements. Structured statements are composed of parts that are themselves statements. They are used to express sequencing and conditional, selective, and repetitive execution. A statement may also be empty, in which case it denotes no action. The empty statement

is included in order to relax punctuation rules in statement sequences.

```
$  statement = [assignment | ProcedureCall |
$         IfStatement | CaseStatement |
$         WhileStatement | RepeatStatement |
$         LoopStatement | WithStatement | EXIT |
$         RETURN [expression] ].
```

9.1. Assignments

The assignment serves to replace the current value of a variable by a new value specified by an expression. The assignment operator is written as ":=" and pronounced as *becomes.*

```
$ assignment = designator ":=" expression.
```

The type of the expression must be included by the type of the variable, or it must extend the type of the variable. The following exceptions hold:

1. The constant NIL can be assigned to variables of any pointer or procedure type.

2. Strings can be assigned to any variable whose type is an array of characters, provided the length of the string is less than that of the array. If a string s of length n is assigned to an array a , the result is a[i] = si for i = 0 ... n-1, and a[n] = 0X.

Examples of assignments (see examples in Ch. 7):

```
i := 0
p := i = j
x := i + 1
k := log2(i+j)
F := log2
s := {2, 3, 5, 7, 11, 13}
a[i] := (x+y) * (x-y)
t.key := i
w[i+1].ch := "A"
```

9.2. Procedure calls

A procedure call serves to activate a procedure. The procedure call may contain a list of actual parameters which are substituted in place of their corresponding formal parameters defined in the procedure declaration (see Ch. 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: *variable* and *value parameters.*

In the case of variable parameters, the actual parameter must be a designator denoting a variable. If it designates an element of a structured variable, the selector is evaluated when the formal/actual parameter substitution takes place, i.e. before the execution of the procedure. If the parameter is a value parameter, the corresponding actual parameter must be an expression. This expression is evaluated prior to the procedure activation, and the resulting value is assigned to the formal parameter which now constitutes a local variable (see also 10.1.).

```
$ ProcedureCall = designator [ActualParameters].
```

Examples of procedure calls:

```
ReadInt(i)        (see Ch. 10)
WriteInt(j*2+1, 6)
INC(w[k].count)
```

9.3. Statement sequences

Statement sequences denote the sequence of actions specified by the component statements which are separated by semicolons.

```
$ StatementSequence = statement {";" statement}.
```

9.4. If statements

```
$ IfStatement = IF expression THEN StatementSequence
$      {ELSIF expression THEN StatementSequence}
$      [ELSE StatementSequence]
$      END.
```

If statements specify the conditional execution of guarded statements. The Boolean expression preceding a statement is called its *guard.* The guards are evaluated in sequence of occurrence, until one evaluates to TRUE, whereafter its associated statement sequence is executed. If no guard is satisfied, the statement sequence following the symbol ELSE is executed, if there is one.

Example:

```
IF (ch >= "A") & (ch <= "Z") THEN
   ReadIdentifier
ELSIF (ch >= "0") & (ch <= "9") THEN
   ReadNumber
ELSIF ch = 22X THEN
   ReadString
END
```

9.5. Case statements

Case statements specify the selection and execution of a statement sequence according to the value of an expression. First the case expression is evaluated, then the statement sequence is executed whose case label list contains the obtained value. The case expression and all labels must be of the same type, which must be an integer type or CHAR. Case labels are constants, and no value must occur more than once. If the value of the expression does not occur as a label of any case, the statement sequence following the symbol ELSE is selected, if there is one. Otherwise it is considered as an error.

```
$ CaseStatement =      CASE expression OF case
$        {"|" case} [ELSE StatementSequence] END.
$ case =         [CaseLabelList ":" StatementSequence].
$ CaseLabelList  = CaseLabels {"," CaseLabels}.
$ CaseLabels  = ConstExpression [".." ConstExpression].
```

Example:

```
CASE ch OF
 "A" .. "Z":    ReadIdentifier
| "0" .. "9":   ReadNumber
| 22X :         ReadString
ELSE            SpecialCharacter
END
```

9.6. While statements

While statements specify repetition. If the Boolean expression (guard) yields TRUE, the statement sequence is executed. The expression evaluation and the statement execution are repeated as long as the Boolean expression yields TRUE.

```
$ WhileStatement  = WHILE expression DO
$        StatementSequence END.
```

Examples:

```
WHILE j > 0 DO
   j := j DIV 2;
   i := i+1
END

WHILE (t # NIL) & (t.key # i) DO
   t := t.left
END
```

9.7. Repeat Statements

A repeat statement specifies the repeated execution of a statement sequence until a condition is satisfied. The statement sequence is executed at least once.

```
$ RepeatStatement  =  REPEAT StatementSequence
$        UNTIL expression.
```

9.8. Loop statements

A loop statement specifies the repeated execution of a statement sequence. It is terminated by the execution of any exit statement within that sequence (see 9.9).

```
$ LoopStatement  =  LOOP StatementSequence END.
```

Example:

```
LOOP
  IF t1 = NIL THEN EXIT END ;
  IF k < t1.key THEN
    t2 := t1.left; p := TRUE
  ELSIF k > t1.key THEN
    t2 := t1.right; p := FALSE
  ELSE EXIT
  END ;
  t1 := t2
END
```

Although while and repeat statements can be expressed by loop statements containing a single exit statement, the use of while and repeat statements is recommended in the most frequently occurring situations, where termination depends on a single condition determined either at the beginning or the end of the repeated statement sequence. The loop statement is useful to express cases with several termination conditions and points.

9.9. Return and exit statements

A return statement consists of the symbol RETURN, possibly followed by an expression. It indicates the termination of a procedure, and the expression specifies the result of a function procedure. Its type must be identical to the result type specified in the procedure heading (see Ch. 10).

Function procedures require the presence of a return statement indicating the result value. There may be several, although only one will be executed. In proper procedures, a return statement is implied by the end of the procedure body. An explicit return statement therefore appears as an additional (probably exceptional) termination point.

An exit statement consists of the symbol EXIT. It specifies termination of the enclosing loop statement and continuation with the statement following that loop state-

ment. Exit statements are contextually, although not syntactically bound to the loop statement which contains them.

9.10. With statements

If a pointer variable or a variable parameter with record structure is of a type T0, it may be designated in the heading of a with clause together with a type T that is an extension of T0. Then the variable is guarded within the with statement as if it had been declared of type T. The with statement assumes a role similar to the type guard, extending the guard over an entire statement sequence. It may be regarded as a *regional type guard.*

```
$ WithStatement  =  WITH qualident ":" qualident
$        DO StatementSequence END .
```

Example:

```
WITH t: CenterNode DO
  name := t.name; L := t.subnode
END
```

10. Procedure declarations

Procedure declarations consist of a *procedure heading* and a *procedure body.* The heading specifies the procedure identifier, the *formal parameters,* and the result type (if any). The body contains declarations and statements. The procedure identifier is repeated at the end of the procedure declaration. There are two kinds of procedures, namely *proper procedures* and *function procedures.* The latter are activated by a function designator as a constituent of an expression, and yield a result that is an operand in the expression. Proper procedures are activated by a procedure call. The function procedure is distinguished in the declaration by indication of the type of its result following the parameter list. Its body must contain a RETURN statement which defines the result of the function procedure.

All constants, variables, types, and procedures declared within a procedure body are *local* to the procedure. The values of local variables are undefined upon entry to the procedure. Since procedures may be declared as local objects too, procedure declarations may be nested.

In addition to its formal parameters and locally declared objects, the objects declared in the environment of the procedure are also visible in the procedure (with the exception of those objects that have the same name as an object declared locally).

The use of the procedure identifier in a call within its declaration implies recursive activation of the procedure.

```
$ ProcedureDeclaration  =  ProcedureHeading ";"
$         ProcedureBody ident.
$ ProcedureHeading  =  PROCEDURE identdef
$         [FormalParameters].
$ ProcedureBody  =  DeclarationSequence
$         [BEGIN StatementSequence] END.
$ ForwardDeclaration  =  PROCEDURE "^" identdef
$         [FormalParameters].
$ DeclarationSequence  =  {CONST {ConstantDeclaration ";"} |
$         TYPE {TypeDeclaration ";"} |
$         VAR {VariableDeclaration ";"}}
$         {ProcedureDeclaration ";" | ForwardDeclaration ";"}.
```

A *forward declaration* serves to allow forward references to a procedure that appears later in the text in full. The actual declaration - which specifies the body - must indicate the same parameters and result type (if any) as the forward declaration, and it must be within the same scope.

10.1. Formal parameters

Formal parameters are identifiers which denote actual parameters specified in the procedure call. The correspondence between formal and actual parameters is established when the procedure is called. There are two kinds of parameters, namely *value* and *variable parameters.* The kind is indicated in the formal parameter list. Value parameters stand for local variables to which the result of the evaluation of the corresponding actual parameter is assigned as initial value. Variable parameters correspond to actual parameters that are variables, and they stand for these variables. Variable parameters are indicated by the symbol VAR, value parameters by the absence of the symbol VAR. A function procedure without parameters must have an empty parameter list. It must be called by a function designator whose actual parameter list is empty too.

Formal parameters are local to the procedure, i.e. their scope is the program text which constitutes the procedure declaration.

```
$ FormalParameters  =  "(" [FPSection {";" FPSection}] ")"
$         [":" qualident].
$ FPSection  =  [VAR] ident  {"," ident} ":" FormalType.
$ FormalType  =  {ARRAY OF}  qualident | ProcedureType.
```

The type of each formal parameter is specified in the parameter list. For variable parameters, it must be identical to the corresponding actual parameter's type, except in the case of a record, where it must be a base type of the corresponding actual parameter's type. For value parameters, the rule of assignment holds (see 9.1). If the formal parameter's type is specified as ARRAY OF T

the parameter is said to be an *open array parameter*, and the corresponding actual parameter may be any array with element type T.

If a formal parameter specifies a procedure type, then the corresponding actual parameter must be either a procedure declared at level 0 or a variable (or parameter) of that procedure type. It cannot be a predefined procedure. The result type of a procedure can be neither a record nor an array.

Examples of procedure declarations:

```
PROCEDURE ReadInt(VAR x: INTEGER);
  VAR i : INTEGER; ch: CHAR;
BEGIN
  i := 0; Read(ch);
  WHILE ("0" <= ch) & (ch <= "9") DO
    i := 10*i + (ORD(ch)-ORD("0"));
    Read(ch)
  END ;
  x := i
END ReadInt
```

```
PROCEDURE WriteInt(x: INTEGER);
                              (* 0 <= x < 10^5 *)
VAR i: INTEGER;
     buf: ARRAY 5 OF INTEGER;
BEGIN i := 0;
 REPEAT
   buf[i] := x MOD 10;
   x := x DIV 10;
   INC(i)
 UNTIL x = 0;
  REPEAT
    DEC(i);
    Write(CHR(buf[i] + ORD("0")))
  UNTIL i = 0
END WriteInt
```

```
PROCEDURE log2(x: INTEGER): INTEGER;
  VAR y: INTEGER;  (*assume x>0*)
BEGIN y := 0;
  WHILE x > 1 DO
    x := x DIV 2;
    INC(y)
  END ;
  RETURN y
END log2
```

10.2. Predefined procedures

The following table lists the predefined procedures. Some are *generic* procedures, i.e. they apply to several types of operands. v stands for a variable, x and n for expressions, and T for a type.

Function procedures:

Name	Argument type	Result type	Function
ABS(x)	numeric type	type of x	absolute value
ODD(x)	integer type	BOOLEAN	x MOD 2 = 1
CAP(x)	CHAR	CHAR	corresponding capital letter
ASH(x, n)	x, n: integer type	LONGINT	x * 2 n , arithmetic shift
LEN(v, n)	v: array n: integer type	LONGINT	the length of v in dimension n
LEN(v)	is equivalent with LEN(v, 0)		
MAX(T)	T = basic type	T	maximum value of type T
	T = SET	INTEGER	maximum element of sets
MIN(T)	T = basic type	T	minimum value of type T
	T = SET	INTEGER	0
SIZE(T)	T = any type	integer type	no. of bytes required by T

Type conversion procedures:

Name	Argument type	Result type	Function
ORD(x)	CHAR	INTEGER	ordinal number of x
CHR(x)	integer type	CHAR	character with ordinal number x
SHORT(x)	LONGINT INTEGER LONGREAL	INTEGER SHORTINT REAL	identity (truncation possible)
LONG(x)	SHORTINT INTEGER REAL	INTEGER LONGINT LONGREAL	identity
ENTIER(x)	real type	LONGINT	largest integer not greater than x

Note that ENTIER(i/j) = i DIV j

Proper procedures:

Name	Argument types	Function
INC(v)	integer type	v := v+1
INC(v, x)	integer type	v := v+x
DEC(v)	integer type	v := v-1
DEC(v, x)	integer type	v := v-x

INCL(v, x)	v: SET; x: integer type	v := v + {x}
EXCL(v, x)	v: SET; x: integer type	v := v - {x}
COPY(x, v)	x: character array, string v: character array	v := x
NEW(v)	pointer type	allocate v^
HALT(x)	integer constant	terminate program execution

The second parameter of INC and DEC may be omitted, in which case its default value is 1. In HALT(x), x is a parameter whose interpretation is left to the underlying system implementation.

11. Modules

A module is a collection of declarations of constants, types, variables, and procedures, and a sequence of statements for the purpose of assigning initial values to the variables. A module typically constitutes a text that is compilable as a unit.

```
$ module = MODULE ident ";" [ImportList]
$         DeclarationSequence [BEGIN StatementSequence]
$         END ident "." .
$ ImportList = IMPORT import {"," import} ";" .
$ import = ident [":=" ident].
```

The import list specifies the modules of which the module is a client. If an identifier x is exported from a module M, and if M is listed in a module's import list, then x is referred to as M.x. If the form "M := M1" is used in the import list, that object declared within M1 is referenced as M.x .

Identifiers that are to be visible in client modules, i.e. outside the declaring module, must be marked by an export mark in their declaration.

The statement sequence following the symbol BEGIN is executed when the module is added to a system (loaded). Individual (parameterless) procedures can thereafter be activated from the system, and these procedures serve as *commands.*

Example:

```
MODULE Out;
  (*exported procedures: Write, WriteInt, WriteLn*)
  IMPORT Texts, Oberon;

  VAR W: Texts.Writer;

  PROCEDURE Write*(ch: CHAR);
  BEGIN
    Texts.Write(W, ch)
  END Write;

  PROCEDURE WriteInt*(x, n: LONGINT);
    VAR i: INTEGER; a: ARRAY 16 OF CHAR;
  BEGIN i := 0;
    IF x < 0 THEN
      Texts.Write(W, "-"); x := -x
    END ;
    REPEAT
      a[i] := CHR(x MOD 10 + ORD("0"));
      x := x DIV 10; INC(i)
    UNTIL x = 0;
    REPEAT
      Texts.Write(W, " ");
      DEC(n)
    UNTIL n <= i;
    REPEAT
      DEC(i);
      Texts.Write(W, a[i])
    UNTIL i = 0
  END WriteInt;

  PROCEDURE WriteLn*;
  BEGIN
    Texts.WriteLn(W);
    Texts.Append(Oberon.Log, W.buf)
  END WriteLn;

BEGIN
  Texts.OpenWriter(W)
END Out.
```

12. The Module SYSTEM

The module SYSTEM contains definitions that are necessary to program *low-level* operations referring directly to resources particular to a given computer and/or implementation. These include for example facilities for accessing devices that are controlled by the computer, and facilities to break the data type compatibility rules otherwise imposed by the language definition. It is recommended to restrict their use to specific *low-level* modules. Such modules are inherently non-portable, but easily recognized due to the identifier SYSTEM appearing in their import lists. The subsequent definitions are applicable to most modern computers; however, individual implementations may include in this module definitions that are particular to the specific, underlying computer.

Module SYSTEM exports the data type BYTE. No representation of values is specified. Instead, certain compatibility rules with other types are given:

1. The type BYTE is compatible with CHAR and SHORTINT.

2. If a formal VAR parameter is of type ARRAY OF BYTE, then the corresponding actual parameter may be of any type.

The procedures contained in module SYSTEM are listed in the following tables. They correspond to single instructions compiled as in-line code. For details, the reader is referred to the processor manual. v stands for a variable, x, y, a, and n for expressions, and T for a type.

Function procedures:

Name	Argument types	Result type	Function
ADR(v)	any	LONGINT	address of variable v
BIT(a, n)	a: LONGINT n: integer type	BOOLEAN	bit n of Mem[a]
CC(n)	integer constant	BOOLEAN	Condition n (0 <= n < 16)
LSH(x, n)	x: integer type or SET n: integer type	type of x	logical shift
ROT(x, n)	x: integer type or SET n: integer type	type of x	rotation
VAL(T, x)	T, x: any type	T	x interpreted as of type T

Proper procedures:

Name	Argument types	Function
GET(a, v)	a: LONGINT; v: any basic type	v := Mem[a]
PUT(a, x)	a: LONGINT; x: any basic type	Mem[a] := x
MOVE(s, d, n)	s, d: LONGINT; n: integer type	Mem[d] ... Mem[d+n-1] :=Mem[s] ... Mem[s+n-1]
NEW(v, n)	v: any pointer type n: integer type	allocate storage block of n bytes assign its address to v

File: OberonReport.Doc / NW 1.10.90

Oberon EBNF

```
ident = letter {letter | digit}.
number = integer | real.
integer = digit {digit} | digit {hexDigit} "H".
real = digit {digit} "." {digit} [ScaleFactor].
ScaleFactor = ("E" | "D") ["+" | "-"] digit {digit}.
hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
CharConstant = '"' character '"' | digit {hexDigit} "X".
string = '"' {character} '"' .

identdef = ident ["*"].
qualident = [ident "."] ident.
ConstantDeclaration = identdef "=" ConstExpression.
ConstExpression = expression.
TypeDeclaration = identdef "=" type.
type = qualident | ArrayType | RecordType | PointerType | ProcedureType.
ArrayType = ARRAY length {"," length} OF type.
length = ConstExpression.
RecordType = RECORD ["(" BaseType ")"] FieldListSequence END.
BaseType = qualident.
FieldListSequence = FieldList {";" FieldList}.
FieldList = [IdentList ":" type].
```

```
IdentList = identdef {"," identdef}.
PointerType = POINTER TO type.
ProcedureType = PROCEDURE [FormalParameters].
VariableDeclaration = IdentList ":" type.

designator = qualident {"." ident | "[" ExpList "]" | "(" qualident ")" | "^" }.
ExpList = expression {"," expression}.
expression = SimpleExpression [relation SimpleExpression].
relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
SimpleExpression = ["+"|"-"] term {AddOperator term}.
AddOperator = "+" | "-" | OR .
term = factor {MulOperator factor}.
MulOperator = "*" | "/" | DIV | MOD | "&" .
factor = number | CharConstant | string | NIL | set |
 designator [ActualParameters] | "(" expression ")" | "~" factor.
set = "{" [element {"," element}] "}".
element = expression [".." expression].
ActualParameters = "(" [ExpList] ")" .
statement = [assignment | ProcedureCall |
 IfStatement | CaseStatement | WhileStatement | RepeatStatement |
 LoopStatement | WithStatement | EXIT | RETURN [expression] ].
assignment = designator ":=" expression.
ProcedureCall = designator [ActualParameters].
StatementSequence = statement {";" statement}.
IfStatement = IF expression THEN StatementSequence
 {ELSIF expression THEN StatementSequence}
 [ELSE StatementSequence] END.
CaseStatement = CASE expression OF case {"|" case}
 [ELSE StatementSequence] END.
case = [CaseLabelList ":" StatementSequence].
CaseLabelList = CaseLabels {"," CaseLabels}.
CaseLabels = ConstExpression [".." ConstExpression].
WhileStatement = WHILE expression DO StatementSequence END.
RepeatStatement = REPEAT StatementSequence UNTIL expression.
LoopStatement = LOOP StatementSequence END.
WithStatement = WITH qualident ":" qualident DO StatementSequence END .

ProcedureDeclaration = ProcedureHeading ";" ProcedureBody ident.
ProcedureHeading = PROCEDURE ["*"] identdef [FormalParameters].
ProcedureBody = DeclarationSequence [BEGIN StatementSequence] END.
ForwardDeclaration = PROCEDURE "^" ident ["*"] [FormalParameters].
DeclarationSequence = {CONST {ConstantDeclaration ";"} |
  TYPE {TypeDeclaration ";"} | VAR {VariableDeclaration ";"}}
  {ProcedureDeclaration ";" | ForwardDeclaration ";"}.
FormalParameters = "(" [FPSection {";" FPSection}] ")" [":" qualident].
FPSection = [VAR] ident {"," ident} ":" FormalType.
FormalType = {ARRAY OF} (qualident | ProcedureType).
ImportList = IMPORT import {"," import} ";" .
import = ident [":=" ident].
module = MODULE ident ";" [ImportList] DeclarationSequence
  [BEGIN StatementSequence] END ident "." .
```

Oberon Availability

Public Domain Oberon (from ETH)

Oberon - the successor of Modula-2 - is both a programming language and an operating system designed by N.Wirth and J.Gutknecht at ETH Zurich. It is available as public domain software from ETH. Currently there are implementations for Apple Macintosh II, Digital Equipment DECstation, and Sun SPARCStation. Implementations for IBM PC (OS2) and IBM RS6000 are under development. The following lists some characteristics of the Oberon system and describes how to get it.

Language
- Strong type checking
- Modules with type checked interfaces and separate compilation
- Type extension
- Support for run-time type tests
- Compatibility between all numeric types (mixed expressions)
- String operations

Compiler
- Generates native code; no separate linking necessary
- Speed: more than 1000 lines per second on a SPARCStation1

System
- Single-process multitasking
- Automatic garbage collection
- Commands: procedures that can be called like programs
- Linking loader
- Dynamic loading (adding modules to a running program)
- Tiling window system
- Text as built-in abstract data type
- Tools for text and graphics editing

Literature

The primary source about the Oberon System, describing the standard module library and how to use the system is the book

M.Reiser: The Oberon System. User Guide and Programmer's Manual. Addison Wesley, 1991, ISBN 0-201-54422-9

Other literature about the Oberon language, about implementation aspects of the system, and about the Oberon System on Macintosh and SparcStation:

- N.Wirth: From Modula to Oberon and The Programming Language Oberon. Software - Practice & Experience, 18, 7 (July 1988)
- N.Wirth, J.Gutknecht: The Oberon System. Software - Practice & Experience, 19, 9 (Sept.1989), 10-18
- M.Franz: MacOberon Reference Manual, Report 142, ETH Zurich, Departement Informatik, 1990
- J.Templ: SPARC-Oberon - User's Guide and Implemen tation. Report 133, ETH Zurich, Departement Infor matik, 1990

Books about the Oberon language and the Oberon project (including main parts of the implementation in source form) are in preparation.

How to get Oberon

Oberon can be obtained via anonymous internet file transfer ftp (at no charge) or on floppy disks (send 20 Swiss Francs or 20 US Dollars to the address below and specify the desired version of Oberon). If you obtain Oberon via ftp, documentation is included in machine-readable form. If you order it on floppy disks, the basic documentation is included in paper.

Hostname: neptune.inf.ethz.ch
Internet Address: 129.132.101.33
Login Name: anonymous
Password: <your e-mail address>
Directory: Oberon (there are subdirectories named MacII,SPARC and DECstation)

For any further questions please contact

ETH Zuerich, Institut fuer Computersysteme (Secretary)
CH-8092 Zuerich
Tel.: +41-1-254 7311
Fax: +41-1-262 3973
Electronic Mail: goerlitz@inf.ethz.ch

OBERON-M(tm) version 1.1 ANNOUNCEMENT

This is to announce the immediate availability of Oberon-M version 1.1.

This package presents the Oberon programming language for the MSDOS environment, on Intel 80x86 processors.

Version 1.1 has some language changes to keep pace with Niklaus Wirth's Oberon revisions (see below). Documentation has been increased, and a mature modular example of Oberon's unique features has been added.

Oberon, as you may already know, is a second-generation language past Pascal with most (maybe all) of the clumsiness of its ancestors removed, with the added power of type extension and object oriented programming features available.

This release contains the following items:

compiler, library modules, updated documention,
language report, new/old examples, utility programs

EXAMPLE FILES: a new, robust set of five modules are included to illustrate Oberon's unique type extension and object oriented programming features. They make a fairly good teaching tool about Oberon's strengths. The well-received Abu program and the original library modules are still present.

HOW TO GET Oberon-M version 1.1

The locations below have the new Oberon-M package. It can be obtained by anonymous FTP, or by mail-message glue-and-uudecode, depending on the location.

FTP formats

The package file name on an FTP location is either

oberonm - self unzipping .EXE file
oberonmz - zip files (not self-unzipping)

Either must be fetched using FTP in BINARY mode. Once downloaded to an MSDOS machine, put the following file name extensions on the respective file you obtained:

oberonm ---> oberonm.exe (run it to unzip)
oberonmz ---> oberonmz.zip (use PKZIP on it)

The oberonmz form is provided for those distribution locations that prefer zip data files versus executable ones. Both files otherwise are identical.

MAIL MESSAGE FORMATS

For locations that have the package in mail-like ASCII files (6 files), you must use an editor to concatenate all the files where shown, use uudecode to bring it back into a binary image, then download and use PKZIP (or an equivalent utility) to unpack all the files of the Oberon-M package. Only the zip-data form of the package is included in the mail-like format (ie: it is the oberonmz file mentioned above).

LOCATIONS TO OBTAIN THE PACKAGE

1) SIMTEL20
machine name: WSMR-SIMTEL20.ARMY.MIL
Internet address: 26.2.0.74, 192.88.110.20
subdirectory: pd1:<msdos.pgmutl>
file names: OBRONM11.ZIP
(* NOTE special form of name here*)
fetch how: anonymous FTP
unpack how: PKZIP under MSDOS

2) UCSD
machine name: ucsd, ucsd.edu, pop.ucsd.edu
Internet address: 128.54.16.1
subdirectory: pub
file names: oberonm.exe
fetch how: anonymous FTP
unpack how: For oberonm.exe: binary transfer
to MSDOS, then execute (self unzipping files)

3) ETH Zurich
machine name: neptune.inf.ethz.ch
Internet address: 129.132.101.33
subdirectory: Oberon/80186
file names: oberonm.exe, oberonm.info
fetch how: anonymous FTP

*** Note: ETH has not tested this package extensively and does not claim or disclaim its validity relative to the ETH Oberon System. Keeping the files here is only being done as a courtesy to European users who want to fetch it from a closer location.

4) comp.binaries.ibm.pc
machine name: Usenet newsgroup
Internet address: N/A
file formats: 6 uuencoded "mail" messages
fetch how: capture the messages
unpack how: Instructions are at the
head of the first message.

5) alt.sources
(same as in comp.binaries.ibm.pc , but available here also by request from many users)

6) wuarchive
machine name: wuarchive.wustl.edu
Internet address: 128.252.135.4
subdirectory: /mirrors/msdos/pgmutl
file names: obronm11.zip
(* NOTE special form of name here*)
fetch how: anonymous FTP
unpack how: binary transfer to MSDOS,
then unzip/decompress using PKZIP or equivalent.

7) University of Ulm
machine name: titania.mathematik.uni-ulm.de
Internet address: 134.60.66.21
subdirectory: soft/oberon/oberonm
file names: oberonm.exe, oberonm.info
fetch how: anonymous FTP
unpack how: For oberonm.exe: binary transfer
to MSDOS, then execute (self unzipping files)
For oberonm.info: ascii transfer for human reading

-- E. R. Videki
erv@k2.everest.tandem.com
IP address 130.252.59.153

Oberon for the Amiga

A version of Oberon is available for the Amiga running AmigaDOS. Information on it may be obtained by writing to:

A+L AG Im Däderiz 61 CH-2540 Grenchen Switzerland	Or:	Tenera Merx Productions Pty Ltd Unit 1, 25 Buckingham Drive Wangara WA 6065

Copies of Oberon-M for MSDOS, and a demo version of the Amiga Oberon, are available in lib 3 of the USUS forum on Compuserve. The forum (formerly called MUSUS), may be reached by GO CODEPORT, or GO MODULA.

Special thanks to Nicklaus Wirth for giving permission to reprint the 2 papers on Oberon.

Interupt Routines in JPI Modula-2

by Mike Hughes

I had earlier asked some questions about writing interrupt driven routines in JPI Modula-2. Here are the answers I finally came up with and which seem to be correct in practice.

1. The module "priority" is a bit mask representing the interrupt controller chip's mask register. Bit zero controls IRQ 0, etc. Setting the bit to a one DISABLES the corresponding IRQ when the module is entered. I suspect that this is really implemented only in IOTRANSFER, although it should be a part of the entry and exit code of every routine in the module. Setting the word to FFFF will block all interrupts during the execution of the routine. A value of 18H will disable IRQ 3 and 4, both serial ports.

2. The "Interrupt Vector", which is a parameter in IOTRANSFER, refers to what the BIOS manuals usually call the "Interrupt Number". This is an index into the jump table located at memory address zero, and has values such as 0CH for COM1, 0BH for COM2, etc. Most books on the BIOS or DOS have lists of these things.

3. IOTRANSFER does not un-mask the IRQ associated with the interrupt vector. This is unfortunate since the routine assumes the normal relationship between them for other purposes, and could have done this for us as well. Oh well, maybe version 3...

To do this, create a bit mask with a zero bit only in the position corresponding to the IRQ you want to enable. AND this mask with the current contents of the Interrupt Mask Register located at port address 21H. Save the old value of the mask to restore before leaving the module. It is a good idea to disable the interrupts while modifying the mask to avoid possible weirdness. The procedure is roughly as follows:

```
TYPE BYTESET : SET OF SHORTCARD[0..7];
  (* 8 bit equivalent of BITSET *)
VAR Mask, OldIMR : BYTESET;

Mask := BYTESET{0..3,5..7};
     (* Enable IRQ 4 (COM1) *)
SYSTEM.DI;
OldIMR := BYTESET(SYSTEM.In(21H));
SYSTEM.Out(21H,SHORTCARD(OldIMR*Mask));
SYSTEM.EI;
```

The original mask should be restored before leaving the program:

```
SYSTEM.Out(21H,SHORTCARD(OldIMR);
```

4. The compiler does not generate code to restore the interrupt vector to its original address before leaving the program. If you execute an IOTRANSFER you must do this or your system will crash the next time something happens on that IRQ!

```
VAR
  Vectors [0:0] : ARRAY [0..255] OF
                       LONGCARD;
  SaveVector : LONGCARD;

  SaveVector := Vectors[InterruptVector];

  Vectors[InterruptVector] := SaveVector;
```

Yes, I know, these things should really be pointers to procedures, but that introduces all sorts of meaningless distinctions and this code is system dependent anyway. So long as it has 32 bits.

5. IOTRANSFER does take care executing the End of Interrupt instruction when given interrupt vectors 08H to 0FH. Praise the Maker!

6. A driver will usually be placed inside a local module to establish the interrupt priority and will have the following form:

```
MODULE InterruptModule[priority];

    IMPORT
      (* All variables required
         from outer module *)

    EXPORT Driver;

    PROCEDURE Driver;
     BEGIN
       LOOP
         IOTRANSFER(DriverProcess,
                    SourceProcess,
                    InterruptVector);
         (* Interrupt handling code here *)
       END
     END Driver;
END InterruptModule;
```

After hardware initialization:

```
NEWPROCESS(Driver, ADR(DriverStack),
   SIZE(DriverStack), DriverProcess);
TRANSFER(SourceProcess, DriverProcess);
```

Note that the TRANSFER causes only the IOTRANSFER to be executed in the driver routine. This returns control back to the initialization routine, but sets up the interrupt trap. Entry to the driver will then only occur through the interrupt. If you have separate control of the hardware interrupt (as with the Interrupt Enable register in a serial port chip), this should be enabled as the very last thing.

Also note that the Version 2 compiler requires a FarADR as the second parameter of NEWPROCESS.

Best of luck to anyone else quixotic enough to want to do this stuff!

OPEN SYSTEMS NEWS

UPDATED FEBRUARY 1991 NEWSLETTER

A publication of Pecan Software Europe Ltd Spring 91

NEW 32 BIT VLM FOR MS-DOS AND UNIX

THE new 32 Bit VLM (Very Large machines) Power System is now available and is currently helping many clients to overcome the limitation of 64K of Stack/Heap.

The standard 32 Bit VLM can be configurated from 256K Stack/Heap area to many Megabytes of Stack/Heap memory space.

The MS-DOS version of 32 Bit VLM is currently restricted to 640K RAM, however a version taking advantage of the Intel 80386/80486 and memory extensions is under final development.

The SCO UNIX version of VLM for Intel based computers will also utilise the same memory software technology to address machines with large programming memory and also incorporate window functions.

Ring now for the latest product details and price list.

OPEN UNIVERSITY

An additional 20,000 units of Pecan Power System software, with a retail value of £1.4 million, have been purchased by the Open University for operation during the 1991 and 1992 academic years.

The decision by the Open University to apply a contract purchase option for additional quantities is based on the experience, quality and excellent operational results of the software during the initial academic years.

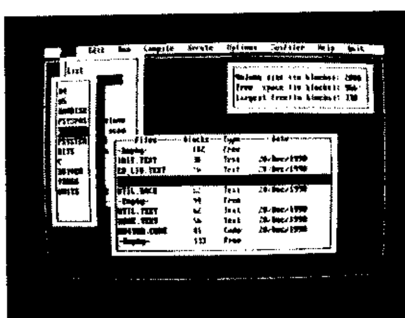
The contract was won by Pecan despite competition from a number of major Pascal vendors.

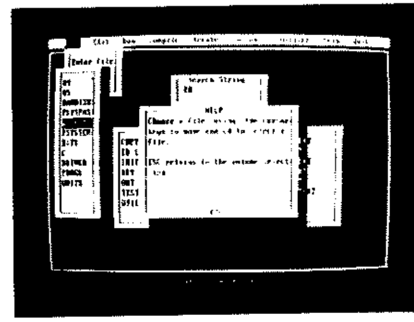
An important factor in the decision making to continue with the Power system is that there is no record of product failure being reported by any of the 12,000 plus students.

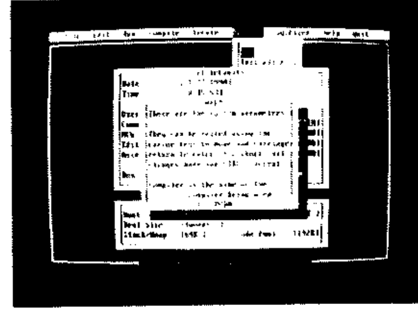
Ken Helps, Managing Director of Pecan Software Europe Ltd says,"It's very satisfying to help contribute significantly to the raising of computing standards and to receive confirmation that the Power system has proved itself through the use of thousands of students".

"The UCSD Pascal programming course is the most popular 2nd year course at the University", states Gordon Davies acting Head of Computing at the University.

THE NEW POWER SYSTEM INTERFACE







The New Power System User Interface as indicated above supports pop-up, pull down menus, help screens, windowing plus much more.

Upgrade to the new MS-DOS version of the User/Interface for only £100 + VAT, p+p.

MSA customers will be supplied free of charge.

We will even include the pop-up library routines and a demonstration disk with source code free of charge.

NEW PRODUCT RELEASES

SCO XENIX/UNIX VERSION OF UCSD PASCAL

The Unix marketplace is growing by 45% p.a., you can now enter that marketplace with your UCSD based software.
See inside pages.

OS/2 VERSION OF UCSD PASCAL

Yes another new product and platform for all UCSD based software to be ported and running within hours. See inside pages.

PASCAL LIBERATOR

Move your Turbo Pascal programs to SCO XENIX/UNIX quickly and easily using UCSD Pascal and our new liberator product.
See inside pages.

UCSD INSTALLATION PROGRAM

At last the first easy to use install program for the UCSD development system available for MSA customers. See inside pages.

SWAPPING COMPILER

A version of UCSD Pascal designed specifically for the writing of very large programs and with increased functions. See inside pages.

PC-CHECKDISK / VIRUS DETECTOR

This product is designed for the non-computer person to help detect general Viruses and as a preventive maintenance system for the actual hard disks in computers and all written in UCSD Pascal. See inside pages.

NEW SERVICES

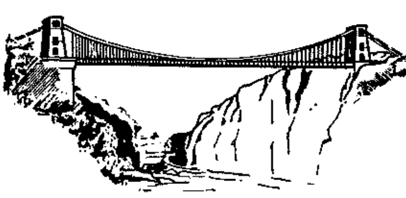
BULLETIN BOARD

News, views, information, upgrade/details, sample programs, programming hints, all provided free of charge, (*ED. does the boss know we're giving this away*).

PORTING CENTRE

For those clients wishing to port UCSD software to new hardware platforms/operating systems and require help and assistance, come on down to Pecan's offices, details inside page.

PECAN EXPANSION



The famous Clifton suspension bridge located near the centre of Bristol and built by Isambard Brunel

Increased office space, new people, plus new products are but part of the planned expansion of Pecan.

Pecan's offices are located in the south west part of the UK in Bristol which is 1.5 hours travelling time to London. Our offices are 10 mins from both the M5 & M4 motorways,with excellent railway and airport links. Bristol is a high tech city and is famous for designing and building the Concorde,the Inmos transputer and historically famous for tobacco, wine and the slave trade. *(so what's changed boss)*

people

New names at the Bristol office

Matthew Porton
He is at present an Undergraduate at Bristol Polytechnic studying for a degree in Systems Analysis and enjoying the experience and benefits of industrial placement for one year. His technical expertise in the application of software coupled with personal interest in total marketing is an asset.

Duncan Rooth
He is a graduate of Cardiff University in the discipline of Computer Science and has industrial/commercial application experience. This experience has allowed Duncan to become proficient in both 'C' and Pascal programming at an advanced level and he is applying his full knowledge as a member of the development team.

Ron Greenaway
He is an experienced businessman, having worked with Multi-national organisations at senior management and director level for many years. Contributions to the many aspects of marketing and administrative control mechanisms are being implemented with resultant benefits to business and customer growth.

THE OPEN SYSTEMS PORTABILITY DEBATE

The discussions of software portability via intermediate formats to produce shrink wrapped software has only just started with a number of offerings, namely -

Unix, 4GL based products and ABI; the world at the moment is ignorant of UCSD Power System, but watch this space.

We would welcome customer opinions of these offerings to achieve portable software, our own views are as follows :-

UNIX

Unix, is becoming the multi-user standard after at least 10 years of hard selling and many millions of £'s spent on hype, and yet Unix versions are still incompatible and many articles have been written endorsing the true nature of Unix and not the message on their advertisements. We find it a great source of amusement collecting the wisdom and comments written about Unix, of course we have a Unix based product already, so true software compatibility is offered under the Unix hosted version of The UCSD Power System.

4GL CASE TOOLS

The theory being that the developer uses one of the many 4GL products, having produced the product he can port data and file structure to other computers quickly.

But alas the dreaded journalists are penning their views of poor productivity with increased program errors to name but a few problem areas.

A recent survey conducted at IBM sites in Holland rejected the adoption of 4GLs as a de facto standard development tool.

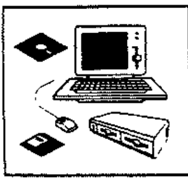
The concept is reasonably good, one of the main problems is lack of any industry standards, each 4GL offering, has it's own command structure and functions which are totally different and incompatible with other 4GL products, also the developer is limited to specific hardware.

ABI

A.B.I Application Binary Interface a good concept of portable software committed to specific families of processors with no compatibility between each family of processors ie Intel - Motorola - Risc and at the moment still a specification not a working product.

When and if the product reaches the market and is accepted we will actually utilise the technology for future products, producing even greater portability of UCSD based software.

PECAN JOINS IBM IN OFFERING PORTING SERVICES.



Yet another new service from Pecan helps software developers port their programs to different hardware and operating systems e.g. Pecans Unix based Power System.

We have a wide range of machines in house. "Our Customers can now familiarise themselves without the setup costs", states Ken Helps. The service is aimed at providing the resources and knowledge to transfer programs quickly, therefore opening yet another hardware market for developers. All at a cost of £250 per day.

Blyth Software's Omnis 3 which is (approx) a 50,000 line UCSD Pascal program was recently ported from a stride environment to an NCR Tower running UNIX in one hour, and the program was fully functional.

PECANS BULLETIN BOARD

A Bulletin Board is a way of transferring information and files from one computer to another using the standard telephone line, a modem and a piece of Comms software, such as the shareware program PROCOMM.

At Pecan Software Europe Ltd we have recently opened our own board, (sometimes called a BBS) to allow our customers and people interested in the Power System to have access to up-to-date information and products 24 hours a day. We also have a number of interesting files from MS-DOS utilities to UCSD Pascal programs that are available for downloading. The Main menu consists of various choices such as Help Level, Questionnaire, etc. As well as these choices, there is also access to further menus, such as the Message menu, File Menu ,Bulletin Menu.

The Bulletin Board number is 0272 248076. We currently support V21, V22 & V22bis speeds, and the file transfer protocols we support are ASCII, XModem CRC, XModem-1K, YModem Batch, ZModem, Kermit, Sealink and Megalink.

POWER SYSTEM UPGRADE VERSION IV 3.0

The new Power System Upgrade is full of new features such as:-

- REVAMPED DYNAMIC SEGMENT MANAGEMENT. Allows double the number of Segments
- 43/50 LINE DISPLAY ON VGA/EGA
- 64 VIRTUAL VOLUMES CAN BE MOUNTED
- NATIVE CODED PASCAL COMPILER. Can double compilation speed
- NEW INSTALL PROGRAM

Don't delay Upgrade today!!

NEW INSTALL PROGRAM

This makes the initial setup of the Power System far easier and faster than ever before. It creates the volumes necessary for the Power System to run your hard disc drive, and then copies the Power System files into these volumes.

Accompanying the program is a short set of instructions, although on screen instructions are produced at the necessary times.

The program is free (except for postage) to those users who have a valid Maintenance and Service Agreement as part of the upgrade to IV.3, simply send us your old discs and we will send you updates by return post.

PASCAL LIBERATOR

Moving your MS-DOS based Turbo Pascal programs to UNIX quickly and easily with our new innovatory Pascal Liberator product.

- Converts Turbo Pascal code to UCSD Pascal running under SCO Xenix/Unix, or any of the following MS-DOS, CPM, OS/2, APPLE the choice is yours.
- Pascal Liberator is designed with a friendly user interface, pop-up menus, help screens and split screen windowing to display differences between Turbo Pascal and UCSD Pascal command structures.
- UCSD Pascal produces an intermediate format and portable programming code.

Liberate your software for only £230

NEW SCO XENIX/UNIX

The Power System supporting UCSD Pascal is now available hosted under SCO Xenix or SCO Unix for IBM based 80286, 80386SX, 80386, 80486 or compatible computers.

Portability

16 bit source and object code developed under Pecan product is upward compatible to the SCO Xenix/Unix versions. UCSD Pascal or any other Pecan language produces P-Code, an intermediate format which allows the developer to port programs to other machines in a very cost effective process. 50,000 line application programs are transferred and operating on the target computer in just a couple of hours.
Alternative methods of porting software would require a complete rewrite usually taking 3/6 months of time and substantial cost.

Development Environment

The UCSD Power System integrated development environment working under SCO Xenix or Unix is identical to all other versions currently available.

Tech Spec/Benefits

Available on all IBM PC and PS/2's and compatibles, running SCO Unix or SCO Xenix. Runs under 80286, 80386, 80386SX 80486 processors, we advise clients to opt for 80386/80486 based development hardware.

Unix Access

These are units to process Unix files, work with Unix pipes, check user names, tty names etc.

Files can be transferred from Power System volumes to Unix and vice versa. One option available even allows you to include your own Unix 'C' code.

Pricing

The professional development system with UCSD Pascal cost £1,200, licensing options are available which include 12 months support free upgrades and preferential pricing for future purchases are available at £3,000.

OS/2 POWER SYSTEM

The OS/2 Power System was developed from the successful SCO Unix/Xenix Power System. It shares many of the same features including: access to OS/2 files, printing via OS/2 spooler, the ability to incorporate your own 'C' code into the Bios (to access Presentation Manager routines for example). The interpreter is written in assembler for maximum performance and the screen updating methods are designed for maximum performance.There is of course full portability of code and data from all our other Power System implementations. This is the quick, easy and efficient way to tap into the growing OS/2 market.

SWAPPING PASCAL COMPILER

This version of UCSD Pascal keeps the symbol table memory-resident as long as possible, for speed, but swaps it to disk when memory is full, this overcomes problems associated with stack overflows.The swapping compiler does not replace the existing compiler, since it's larger therefore compiler speed is reduced, and is offered as a separate product for clients with a Power development system.

PC-CHECKDISK/ VIRUS DETECTOR

PC-Checkdisk/Virus Detector is an easy-to-use tool for monitoring hard disk performance and detecting viruses returning easy to understand error messages in the case that something was wrong.

Performance Testing
The four automated tests are : Disk efficiency, Rotation speed, Verify, and Virus detection.The menu driven software can be used by anybody who can switch on a computer.

The results of the performance test are saved to disk and automatically compared against previous test results, differences between tests are highlighted via warning messages and a recommended course of action.

The product is now complete and has been discussed on both local radio and BBC World Service. One final point is that it is slightly quicker than Nortons DT which is written in 'C' Assembler.

PORTABILITY

GOLD AWARD
FOR UCSD SOFTWARE HOUSE

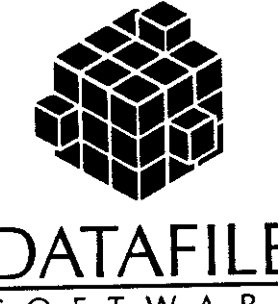
Datafile the Liverpool based software house have recently been accredited with the GOLD AWARD for their accounting software by the leading UK computer publication Micro Decision.

Datafile products were judged superior to Pegasus, Sage, Tetra *- plus many more established names.*

Steve Ashcroft, developer of Datafile programs was happy to say that *"This proves that application software written in UCSD Pascal can be the best"*. Steve also states that *"the portability of UCSD programming code has helped keep us ahead of the competition and saved time on porting software to other systems. This saving of time and cost has been spent on improving our products"*.

Datafile have a complete range of powerful and flexible management accounting software and a number of database products.

We are pleased to be associated with Datafile, a very valued customer and offer our congratulations.



DATAFILE
SOFTWARE

Datafile Software Limited
Brunswick Enterprise Centre
Brunswick Way
Liverpool L3 4BD
Tel: 051 709 0929
Fax: 051 709 2070

CONTACTS AT PECAN (BRISTOL)

The following personnel are pleased to help you with your enquiries:-

Sales: Ken Helps
Matthew Porton

Technical: Gordon Wilkie
Adam Stevens

USUS 10th Anniversary Conference

USUS (UK) Ltd, the UCSD Power System Users' Society, will be holding their 10th anniversary conference at Lancaster University on Friday 12th and Saturday 13th April 1991. Both members and non-members from all European and Commonwealth countries are welcome. The theme for the conference is 'Escaping from DOS'. The Friday will be of a commercial nature with presentations about the experiences of the presenters in porting their software to non-DOS platforms, while Saturday will be designed more along the lines of a technical workshop. Attendance can be for either one or both days.

Enquires about the conference or USUS membership should be addressed to:

Administrator
USUS (UK) Ltd
PO BOX 448
Chelmsford
CM2 8QB

TYPES OF MEMBERSHIP:
Individual - £15 p.a.
(non Company address)
Corporate / Institutional - £75 p.a.
(These membership rates are valid until 31.3.91)

RUNTIME PRICING

To utilise the benefits derived from the Power System in your application software a runtime license is required when selling or copying for other computers..

Runtime pricing is very competitive and with quantity discounts royalty costs can be reduced to a few pounds.

SOFTWARE ENGINEERING/CONSULTANCY

Pecan now offers a range of flexible solutions to cater for customer requirements, such as:-

- Bespoke systems and application programming
- Systems Design and Consultancy
- Training Courses
- Performance Program
- Program optimising for performance and speed

YOUR SAY

Editorial articles and news are always welcome, please send to The Editor c/o Pecan Software Europe Ltd.

CHANGE OF ADDRESS

Please notify Pecan of change in address or should you wish to be kept on or taken off the mailing list.

All prices quoted are in UK pounds sterling and are exclusive of post & packaging and local purchase tax (VAT).

All trademarks acknowledged

740LC

Advanced 68030 UNIX Computer

DESCRIPTION

The 740LC is based upon a state of the art, complete, single board 68030 based CPU card. Optimized for the multi-user UNIX environment, the 740LC offers superior price/performance ratio. The single board CPU provides the major features required for the system. An onboard, high speed hardware floating point coprocessor is optionally available. The 68030 CPU running at 50Mhz, combined with up to 32Mb of RAM and up to 48 serial ports provides the power required in today's sophisticated multi-user environments. The built in SCSI interface provides over 1.2Mbytes/sec sustained transfer rate from the hard disks in the UNIX environment. The optional Ethernet/Cheapernet "sky" board, provides for a high performance Ethernet interface.
Tape backup is provided with a 150Mb, or optionally a 600Mb, digitial data cassette drive. In addition to the tape drive, there is room for one full height, or two half height 5-1/4" SCSI peripherials allowing up 1.5Gb of internal storage. The external SCSI connector allows connection to optical drives/jukeboxes, DAT tape drives, and other SCSI devices.

THE MILLENNIUM 740LC

X-WINDOWS

A comprehensive support package for the X11.4 windowing system is provided. This allows multi-user graphics applications such as WYSIWYG word processing and picture databases. Drivers are provided to display picture fields from popular databases such as Unify and FoxBase, as well as "C" library routines to display and print TIFF format files.

FEATURES

- 68030 CPU operating at 50Mhz
- Unix 5.2 compatable operating system
- X-Windows X11.4 support
- Multi-user Graphics
- Up to 32Mb of Parity checked RAM
- Up to 48 serial ports
- Protection hybrids for all serial lines
- Ethernet/Cheapernet option
- Optional 68882 hardware FPU
- 256Kb of on board EPROM
- Battery backed up Real Time Clock
- Environmental sensors for power supply voltages, temperature, and air flow
- CPU is +5v operation only
- Advanced state of the art surface mount construction
- Rugged heavy gage steel chassis
- "Excess Baggage" certified shipping containers

Board Meeting Minutes (March 13, 1991)

By Keith R. Frederick

Minutes of the Board Meeting of USUS, Inc., held in room 1 of the MUSUS forum teleconferencing facility on the CompuServe Information Service, March 13, 1991.

Present at the meeting were:

User	ID Name
72747,3126	Bob Clark (BobC)
72230,1601	Gary Gibb (Gary)
71016,1203	Stephen Pickett (sfbp)
74076,1715	Felix Bearden (felix)
72767,622	Tom Cattrall (TomC)
73447,2754	Henry Baumgarten (Henry)
73760,3521	Keith Frederick (KeithF)
73007,173	William Smith (Wm)
76702,513	Harry Baya (Harry)

The meeting started at 6:37 PM PST. Topics discussed were:

I. Election Results

Gary Gibb, Keith Frederick, and Felix Bearden won the three open Board of Director positions.

The change to the bylaws was passed overwhelmingly and the decision to keep Computer Language Magazine was split roughly even both for and against.

II. Computer Language Magazine (CLM)

Tom Cattrall said with the CLM vote so close he wasn't sure exactly what USUS should do.

Henry Baumgarten asked whether there have been any responses to the advertisement in CLM and Felix Bearden responded that there had been about 50 responses but he hadn't yet sent letters since he was waiting for stationary. Felix then noted that he hadn't sent out renewals yet because of the uncertainty with CLM.

Henry said that something must be done promptly on the 50 requests for information and that a decision on CLM should be made now.

Stephen Pickett indicated that he felt that the ad responses justify our giving CLM the benefit. Felix agreed.

Felix then motioned to "continue with CLM for the fiscal year of 1991." Henry and Stephen Pickett seconded and Henry then called for a discussion. There was none and a vote was called.
William Smith, Felix Bearden, Stephen Pickett, Harry Baya, Tom Cattrall, Gary Gibb, and Keith Frederick all voted in favor with none opposing. The motion passed.

III. Renewals, Inquiries, and New Officers

Henry started by saying that three issues still stand: 1). new officers, 2). getting out renewal letters ASAP, and 3). answering inquiries to the ad ASAP.

Felix Bearden said that as administrator he would get the renewals out and respond to the requests; effectively taking care of 2 and 3.

Tom Cattrall asked which officer are needed, noting that Administrator, Secretary, someone to chair the meetings, and Treasurer seem to be the essentials. Tom then asked Keith Frederick if he would be willing to continue as secretary. Keith said he would.

Henry then questioned whether, according to the bylaws, a BOD member could serve as an officer. William Smith and Felix Bearden both answered yes.

Henry then considered the Treasurer position and asked about Bob Clark's preferences. Tom Cattrall noted that Bob sounded as if planned to continue as Treasurer. [NOTE : Bob Clark had to leave early so he wasn't present to answer] Tom then said that until it is known for sure, that it is best to assume he will continue.

Henry responded saying if there weren't any objections that would be his preference as well. There were no objections and Henry then continued to the position of Administrator.

Felix Bearden said he would continue as long as needed and noted that the job requires more time than he can give it. Felix commented that he had offered a motion to grant the Institution who assumed the duties an institutional membership so a administrator could be recruited but the motion was never acted. Felix noted that he believed the idea was still a good one.

Tom Cattrall stated to Felix that perhaps he could think of some of the duties that could be split between the administrator and another volunteer.

Henry asked for other comments, there were none and Henry continued to the position of Chair of the BoD meeting. Henry started by saying that a President can be elected to preside or one of the BOD members can be chosen to serve as chair and then asked for comments.

Felix stated that a Chairman of the Board be elected since that position is described in the bylaws and is subject to BoD attendance criteria. Felix then moved to elect a Chairman of the Board. Henry asked for comments, there were none and a vote was called.

William Smith, Stephen Pickett, Harry Baya, Stephen Pickett, Gary Gibb, and Keith Frederick all voted in favor, with

none opposing. The motion carried.

Henry asked for nominations. William Smith and Felix Bearden both nominated Tom Cattrall. Tom Cattrall then nominated Stephen Pickett. Stephen Pickett then seconded the nomination of Tom Cattrall.

Henry then asked whether Tom or Stephen are willing to run.

Tom indicated that he didn't feel qualified since he does not know the rules of order and also, he added, thought that Stephen Pickett would do a good job.

Stephen said he would accept only if someone would be President and absorb some of the duties.

Gary then, after some contention whether a second was needed for nominations, seconded Stephen Pickett.

Henry asked for any comments, there were none, and then Henry asked to vote for either Stephen Pickett or Tom Cattrall as Chairman of the Board.

Felix Bearden, Keith Frederick, Stephen Pickett, and William Smith voted for Tom Cattrall.

Tom Cattrall and Gary Gibb voted for Stephen Pickett.

Tom Cattrall was elected as the new Chairman of the Board. Henry asked if Tom would like to elect a President. Tom responded that the discussion of that be tabled until next meeting. Felix seconded. Henry called for a vote.

All voted in favor, with none opposing and that business was tabled.

After brief discussion, the SysOp positions were reconsidered and Harry Baya and Tom Cattrall were selected as primary and secondary SysOps for the coming year.

NEXT MEETING

The Board adjourned at 9:09 PM PST and agreed to meet again at 6:30 PM PST / 7:30 MST / 8:30 CST / 9:30 EST April 10, 1991 in Room 1 of the MUSUS conference facility.

Minutes submitted by: Keith R. Frederick

Board Meeting Minutes (April 10, 1991)

By Keith R. Frederick

Minutes of the Board Meeting of USUS, Inc., held in room 1 of the MUSUS forum teleconferencing facility on the CompuServe Information Service, April 10, 1991.

Present at the meeting were:

User	ID Name
72747,3126	Bob Clark (BobC)
72230,1601	Gary Gibb (Gary)
71016,1203	Stephen Pickett (sfbp)
72767,622	Tom Cattrall (TomC)
73447,2754	Henry Baumgarten (Henry)
73760,3521	Keith Frederick (KeithF)
76702,513	Harry Baya (Harry)

The meeting started at 6:38 PM PST. Topics discussed were:

I. New Officers Needed

Tom Cattrall started off by asking if an attempt should be made to elect new officers. Keith Frederick replied, asking if the members knew that new officers are needed. Tom answered that he doesn't believe so and questioned if the Board should solicit in the next Newsletter.

Stephen Pickett asked what officers are needed. Keith answered, a new secretary wouldn't hurt and Tom Cattrall said a new President is needed and that Felix Bearden would be happy to have help and a successor for administration.

Tom Cattrall, due to the lack of any ideas, suggested putting a notice in the Newsletter and then seeing what happens. Keith Frederick then mentioned that using CodePort (formerly MUSUS) to get the world out would help as well.

Stephen Pickett suggested there be a general and permanent ad in any publication of USUS stating that assistance is needed. Also, he said, that CodePort would probably be more productive due to the lag time in the Newsletter. Stephen then asked what kind of response Felix has received, noting that about 200 people requested information on USUS.

Tom Cattrall replied that with a standing request, people would quickly learn to ignore it. Tom then asked for additional ideas, there were none.

II. Administration Status

Bob Clark initiated by saying that USUS has about $5000 in the bank but that in the last month there was no income

and not much mail processed, at least, Bob said, he didn't receive any reports. Bob continued to say, that unless the current membership is supported USUS won't last too long.

Stephen Pickett and Henry Baumgarten both indicated that were concerned that they just received renewal notices while their membership ran out several months ago. Tom Cattrall responded that the reason for this delay was not lack of administration but rather the issue with Computer Language Magazine and the Journal of Pascal, Ada, and Modula-2. However, Tom did say, that since Felix Bearden has so little free time the administration is a problem.

Bob Clark expressed his concern about the PO Box in La Jolla indicating that Felix called him the day the rent was due to say he received the noticed but hadn't processed the mail for about two weeks. Bob said he told Felix to pay the fee directly and he would be later reimbursed. Bob then commented that the mail is not being processed as it should be under the current system and that while he (Bob) knows Felix is busy, the problem must be solved.

Tom Cattrall said that Felix had given him some material to put in the Newsletter regarding soliciting some help. Tom then asked for further suggestions.

Gary Gibb asked what type of mail is received. Bob Clark said that while he doesn't receive the mail, the mail would consist of any mail sent to USUS. Tom noted that this means renewals, bills, inquiries, junk mail, etc.

Gary then asked if, other than checks, is there anything of importance. Bob Clark answered "yes" since information on membership, the library, and anything else a member may need to know is handled via mail.

Stephen Pickett then indicated that he had several requests for library materials but that none of them had been in the proper format and asked what he should do. Bob Clark answered that the requests should be submitted on the USUS and then the filled form be sent to me [Bob Clark] and you [Stephen Pickett] will be reimbursed according to the payment schedule.

Tom Cattrall then said that older issues of the newsletter had order forms and instructions and noted that it might be time to publish the rates and rules again.

Tom asked for more ideas regarding the administrators function. Gary replied that maybe a paid secretary may be needed to get the job done, but also noted that the funds do not permit this.

After several minutes of no solid ideas regarding the admin position, Tom Cattrall ask for a vote to adjourn the discussion to section 16 over the next couple of weeks. Stephen Pickett moved it. Stephen Pickett, Gary Gibb, Keith Frederick, and Harry Baya (for the record) all voted in favor with none opposing, the vote passed.

Stephen Pickett got in a last word, saying that if there is anything practical that can be done to relieve Felix, it should be done and that perhaps Casey Blank might be asked to start sending mail to one of us for the time being.

NEXT MEETING

The Board adjourned at 8:12 PM PST and agreed to meet again at 6:30 PM PST / 7:30 MST / 8:30 CST / 9:30 EST May 8, 1991 in Room 1 of the MUSUS conference facility.

Minutes submitted by: Keith R. Frederick

Announcements

New Pascal User Group by Doug Chamberlin

I have started a Pascal SIG within the Boston Computer Society. We meet on the third Mondays of each month, except for August. The May 20th meeting will be in Peabody, 15 miles North of Boston. The June and July meetings will be at the BCS IBM PC User Group office/classroom in Newton, MA.

Most of our members use Turbo Pascal, but we are interested in any Pascal versions - P-system, VAX, Unix, Macintosh, etc.

Millennium 740LC

Millennium Computer Corporation announced the new 740LC line of low cost multi-user computer systems. The 740LC is based on a single board 68030 based CPU card running at 25 or 50mhz. Millennium has priced the line to compete with the existing SCO Xenix/Unix PC based platforms, but deliver an "Industrial Strength" Unix to that market.

Optimized for the multi-user Unix environment, the line also runs Unix-hosted p-System, Native p-System and the NSI-DANIEL (Modula-2 based) operating system developed by Nervous Systems, Inc. for high speed data acquisition applications.

Also available in the new Unistride (Unix) version is Mac/PC NFS which can be used as a fileserver application to optimize storage in a networking environment using the 740 for the server.

Treasurer's Report
by Robert E. Clark, Treasurer

February 1991

```
Bank Balance          $5,055.89 1-31-91

Income - February 1991

Dues:
                                (new/renew)
  Student                 0.00     0/0
  General               180.00     1/3
  Professional            0.00     0/0
  Institutional           0.00     0/0

Other Income:
  CIS                    49.34 (2 months)
  Library fees           21.45
  JPAM                   96.00
                       -------

Total Income:      $    346.79 Expenses -
February 1991

  Bank charges            1.76
  Newsletter              0.00
  Mail from La Jolla      0.00
  Refunds                 0.00
  Reimbursements         12.07
                     --------

Total Expenses     $     13.83

Bank Balance       $5,388.85 2-28-91
```

March / April 1991

```
Bank Balance        $5,388.85 2-28-91

Income - March/April 1991

Dues:                             (new/renew)

  Student                   0.00     0/0
  General                   0.00     0/0
  Professional              0.00     0/0
  Institutional             0.00     0/0

Other Income:
  CIS                       0.00
  Library fees              0.00
                        --------
Total Income:       $       0.00


Expenses - March/April 1991

  Bank charges              3.48
  Newsletter              386.65
  Mail from La Jolla        0.00
  Refunds                   0.00
  Reimbursements           85.40
                      ----------
Total Expenses      $     475.53

Bank Balance        $4,913.32 4-30-91
```

Submission Guidelines

Submit articles to me at the address shown on the back cover. Electronic mail is probably best, disks next best, and paper copy is last. If your article has figures or diagrams, I can use encapsulated Postscript files in any of the disk formats listed below. If you can't produce encapsulated Postscript, then paper copy is probably the only practical method for submitting graphics.

You can send E-Mail to my Compuserve ID: 72767,622, or indirectly from internet: 72767.622@compuserve.com. For disks, I can read Sage/Stride/Pinnacle format disks. Also, any MS-DOS 5.25 or 3.5 disks, and 3.5" Amiga disks. If anyone wants to send Mac format disks I could probably get someone to translate them into something I can use. Whatever you send, please mark on the disk what format it is. That will save me a lot of guesswork.
Text should be plain ascii rather than a word processor file. It can have carriage returns at the end of all lines or only at the ends of paragraphs. What you send doesn't have to look pretty. I will take care of that. My spelling checker will take care of spelling errors too. If you want special formatting use the following conventions:

1. _Underline_, put an underline character at each end of the section to <u>underline</u>.

2. *Bold*, put a star at each end of the section to **bold**.

3. ^Italics^, put a caret at each end of the section to be set in *italics*.

4. ??Special requests??, such as ??box next paragraph?? should be surrounded with "?? ??".

NewsLetter Editor : Tom Cattrall
Amity Software Inc.
7600 Seawood Road SE
Amity, Oregon 97101
503/835-1613
Compuserve : 72767,622
Internet : 72767.622@compuserve.com
tomc@techbook.com

NewsLetter Publisher : William Smith

USUS Board of Directors

Felix Bearden	74076,1715
Tom Cattrall	72767,622
Keith Frederick	73760,3521
Gary Gibb	72230,1601
Stephen Pickett	71016,1203

USUS Officers

President:		
Treasurer:	Bob Clark	72747,3126
Secretary:	Keith Frederick	73760,3521

USUS Staff

Administrator:	Keith Frederick	73760,3521
Legal Advisor:	David R. Babb	72257,1162
Librarian:	Stephen Pickett	71016,1203
MUSUS Sysop:	Harry Baya	76702,513

USUS Membership Information

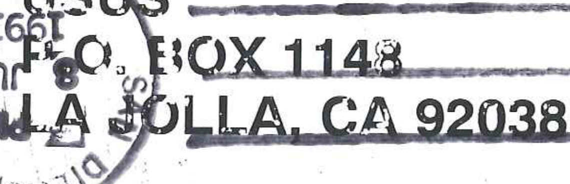
Student Membership	$ 30 / year
Regular Membership	$ 45 / year
Professional Membership	$ 100 / year

$15 special handling outside USA, Canada, and Mexico.

Write to the La Jolla address to obtain a membership form.

NewsLetter Publication Dates

Issue	Due Date For All Newsletter Material
May / Jun 1991	May 1, 1991
Jul / Aug 1991	July 1, 1991
Sep / Oct 1991	September 1, 1991
Nov / Dec 1991	November 1, 1991



ADDRESS CORRECTION REQUESTED
FIRST CLASS MAIL