PUC

Series: Monographs in Computer Science
and Computer Applications
Nº 8/75

ON THE USE OF POINTERS AND THE TEACHING OF DISCIPLINED PROGRAMMING

by

Miguel Angelo A. Nôvoa

and

Sergio Carvalho

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

Series: Monographs in Computer Science
and Computer Applications
Nº 8/75

ON THE USE OF POINTERS AND THE TEACHING OF DISCIPLINED PROGRAMMING*

by

Miguel Angelo A. Nõvoa

and

Sergio Carvalho

Series Editor: Larry Kerschberg

December, 1975

* This work was supported in part by the Brazilian Government Agency FINEP under contract Nº 244/CT, and developed by the Information Systems Group, Departamento de Informática, PUC/RJ.

ABSTRACT

In the past few years there has been considerable debate over the question of pointers in programming languages. Some maintain that pointers should not be allowed , while others try to restrict their use in a number of ways. In this paper we try to justify our view that pointers are a natural and useful way to teach beginners in Computer Science to manipulate list structures, provided a group of strong limitations is placed upon them. We define and give the implementation model of the use of pointers in SPL, a language to teach beginners disciplined programming.

KEY WORDS

Disciplined programming, programming teaching, pointer, record, programming language, type violation, dangling reference.

RESUMO

O uso de variáveis apontadoras em linguagens de programação tem sido ultimamente um assunto bastante discutido. Alguns acham que variáveis deste tipo não deveriam estar presentes em linguagens de programação, enquanto outros preferem restringir o seu uso de várias maneiras. Nesta monografia tentamos justificar o uso de variáveis apontadoras como uma ferramenta natural e útil ao ensino de manipulação de listas para principiantes em computação, desde que estas variáveis sejam submetidas a um rigoroso controle. Esta monografia contém a definição e o modelo de implementação de variáveis apontadoras em SPL, uma linguagem desenvolvida para o ensino de programação estruturada a principiantes.

PALAVRAS CHAVE

Programação estruturada, ensino de programação, linguagem de programação, verificação de tipo, variáveis apontadoras, estruturas de dados.

CONTENTS

1. INTRODUCTION

1.1. THE SPL PROJECT

In 1974, while teaching a course on Data Structures, one of the authors felt the need for the design of yet another high level programming language. One of the main objectives of that course was the teaching of disciplined (top-down) programming [WIR 71a] . Among the programming languages available in the system, PL/1 was the one initially chosen to be used throughout the course, due to the inadequacy of all the others. Soon it was clear that this was a poor choice, since most of the students had a fair knowledge of PL/1 and an excellent knowledge of tricky programming in PL/1. It was felt that:

(i) the teaching of top-down program development would be frequently interrupted by students who would already know "the best way to code the solution to that problem in PL/1", thus delegating the task of disciplined programming to the background, in favour of the simpler task of coding;

(ii) top-down structured programming should be introduced with the aid of simple control structures (compound statements, while statements and if-then-else statements); the knowledge of more powerful control structures in PL/1 (such as ON conditions) would be harmful to that presentation.

Some PL/1 characteristics (type conversions, the absence of a "case" construct, the undisciplined use of pointers [HOL 72, ZEL 74], for example) finally led us to the development of a new programming language , SPL (Simple Programming Language).

The SPL system was created to be used as a tool in the introductory teaching of programming languages and techniques. Its main objectives are:

- to be adequate in the teaching of disciplined programming;

- to allow efficient (diagnostic type) compilation;

- to be as simple as possible, yet presenting adequate control and data structuring facilities.

In this paper we present the SPL solution to the use of pointers in this class of high level programming languages. In the remainder of the Introduction we quickly survey pointers and their problems.

In section 2 facilities for the creation and manipulation of structured data in programming languages are described. It is in this context that pointers are mainly used.

The SPL model is presented in section 3. The need for pointers in introductory programming languages is discussed in 3.1, where pointers shall be related to records (structured data types consisting of one or more elements, where the element types are not required to be identical). In 3.2 the syntax and semantics of records and pointers is described. Section 3.3 presents the operations allowed on records and pointers. Finally in section 3.4 the implementation of pointers in SPL is discussed.

In section 4 we conclude this report. The syntax of SPL is presented in the appendix.

1.2. <u>BRIEF COMMENTS ON POINTERS AND THEIR USES</u>

The pointer type is a part of the repertoire of many high level programming languages today: some examples are PL/1, Algol W [WH 66], and Pascal [WIR 71b]. In this section we briefly examine pointer types in those languages and some current opinions regarding the use of pointers.

The use of pointers in PL/1 may give rise to serious type violations. Due to the existence of the function ADDR [ZEL 74], a pointer which is declared to point to a variable of some type can be set to point, via the ADDR function, to a variable of a different type.

Another problem is the dangling reference problem [CHI 73]. A dangling reference occurs when the storage allocated for a variable local to a block (but being pointed at by a pointer global to the block) is deallocated upon block exit. The global pointer is left indicating deallocated area.

Pointers in Algol W (called references) have less flexibility than in PL/1. They were introduced in the language as a mechanism to link instances of records. Furthermore, since records in a program can have widely different structures, reference variables are declared to point to classes of records (a set of instances of records with identical structures). Thus type checking for pointers can be efficiently implemented.

The fact that in Algol W there is no function similar to PL/1's ADDR, and the fact that in the declaration of references the records classes to which they are allowed to refer must be explicitly stated, prevent the occurrence of the dangling reference problem.

In Pascal pointers are used to refer not only to instances of records, but also to (dynamically generated) variables of any other type. However, unlike PL/1, pointers are bound to sets of values of the same type, thus allowing efficient type checking.

Recently, the use of pointers in high level programming languages has been under considerable debate. Hoare [HOA 73_a] maintains that pointers are low level constructs, and, as such, should not be present in high level programming languages ("Their introduction into high level languages has been a step backward from which we may never recover.").

Problems like type violations and dangling references led some to maintain that pointer types should carry the type of the object pointed at, as in Algol 68 [vWn 69].

Although pointers are traditionally considered as a valuable aid in structuring data, Hoare has shown [HOA 73_b, HOA 75] that one can actually construct complex structured types without using pointers.

More recently, Berry [BER 75] has surveyed the pointer problem, suggesting as a compromise solution the use of pointers in connection with the very high level concepts of abstract data types [LZ 74].

2. FACILITIES FOR THE CONSTRUCTION AND MANIPULATION OF STRUCTURED DATA

2.1. DESCRIPTION

There are several ways to structure data in programming languages; a very thorough survey of such methods can be found in [HOA 72] . One such method is characterized by the structured type "record". The following facilities are required for the creation and manipulation of records in programming languages:

(i) a way to describe the elements (fields, nodes) of a record structure. As mentioned before, the element types of a record structure do not have to be identical. Since memory allocation for record type variables is done dynamically, a record description can be considered as a description of memory sections which, at run time, are associated with the record;

(ii) a way to allocate memory for record type variables and associate it to record definitions. Such allocation is done at run time; records are structures which may grow at random, and as such allocation can not be done at compilation time;

(iii) a way to free memory which has become unnecessary for program execution. This permits an efficient use of memory. In general the allocating and freeing of memory involves the use of algorithms for garbage collection;

(iv) once memory positions are allocated for a record type variable it is important to know where those positions are in memory. Such information is normally conveyed by pointer type variables. These are variables whose possible values are addresses in memory which can hold instances of record variables. With the help of pointer variables, access to fields of an instance of a record variable can be done, for instance , by specifying:

1. the name of a pointer to the record variable instance;
2. the name of the record type variable;
3. the name of the particular field wanted;

(v) a way to specify that record fields can contain references to other record instances; in other words, it should be possible to declare record fields of type pointer. This is important for data structuring, since by linking together instances of records one can construct representations of abstract structures such as stacks, queues, binary trees, and so on.

The mechanisms above are basic for the run-time creation and manipulation of records. In the next section we discuss their implementation.

2.2. IMPLEMENTATION

There are two main (and interconnected) decisions to be taken regarding pointer implementation: we must choose a storage scheme, and we must decide when, during the compilation process, the binding of pointers and the structures pointed at is to take place.

The possible storage schemes are basically two. Either we give the programmer the ability to allocate and deallocate storage (as in PL/1) or we implement the concept of retention [BER 70]. In the latter alternative, memory positions corresponding to a structure being pointed at remain accessible until all references to those positions are eliminated; then they are returned to a list of available space. This scheme calls for the existence of a table of pointers, and a mechanism for continuously (and at run time) updating pointer values, thus being very expensive.

On the other hand, we feel that a system in which the programmer has the use of a pair of functions like ALLOCATE and FREE, plus the ability to choose the binding time, is not compatible with the purposes of SPL, being an invitation to undisciplined programming. The solution found has to do with the binding time.

Pointers can be bound to structures either at compilation or at run time. The binding of pointers that are not "typed" (declared independently of structures) can be done only at run time. This again involves the use of a pointer table maintained during the execution of a program, thus causing some heavy overhead. However, if we can tie pointers to structures at compilation time, then no dynamic tables have to be maintained, type checking can be efficiently done, and a more disciplined use of pointers is achieved.

Syntactically typed pointers are present in languages as Algol W, Pascal, Algol 68, among others. In this scheme pointer types are declared together with the types of objects they are allowed to point to.

3. RECORDS AND POINTERS IN SPL

3.1. GENERALITIES

One of the most important course for beginners in Computer Science is a course on Data Structuring. In it some abstract data types, such as stacks, trees and linked lists are introduced. To represent abstract data types, programming languages provide users with an assortment of built-in types. For example, one dimensional arrays can be used to model stacks, with the help of an integer variable that keeps track of the stack top.

Records and pointers were introduced in SPL to allow users to more naturally represent linked lists. List elements are thus represented by record instances, having one or more fields of type pointer, which provide the linking mechanism. The main characteristics of the SPL system, with respect to records and pointers, are:

(i) compile-time binding of pointers to record classes, as in Algol W;

(ii) pointer type fields of a record are allowed only to point to instances of variables declared of the same record type;

(iii) possibility of user's allocation and deallocation;

(iv) SPL is block structured, with automatic deallocation on block exit, and standard scope rules.

Adoption of (i) above saves overhead in compiling, as seen before. (i) and (ii) provide efficient type checking. The main consequence of restriction (ii) is that lists in SPL are homogeneous. We feel this is not a serious restriction, considering the purpose of SPL.

SPL users have the ability to allocate and deallocate record instances; dynamic storage management in SPL is mainly programmer's responsibility. It is our opinion that, due to (i) and (ii) above, SPL is an adequate environment for the teaching of list manipulation.

In addition, since standard scope rules apply, all pointers declared in a block are destroyed upon block exit. It is the programmer's task to ensure that no instances of records declared globally to the block being exited are left without accessing paths. It should be noted that the sintax of SPL prevents pointers from being declared in a scope wider than that of the corresponding record class.

3.2. THE DEFINITION OF RECORDS AND POINTERS

In this section the syntax and semantics of records and pointers in SPL is presented. More about SPL's syntax can be found in the Appendix. In the syntax rules that follow, lower case strings of letters, with possibly one or more underscores, are considered to be nonterminals; strings of capital letters represent reserved words in the system. Square brackets denote that the enclosed sequence of symbols may or may not be present, curly brackets followed by an asterisk denote the occurrence of zero or more instances of the enclosed sequence of symbols.

Records are defined as follows.

```
record-type  →  RECORD (field-list)
field-list   →  field-name-list : field-type

                { ; field-name-list : field-type }*

field-name-list → identifier-list
field-type → simple-type

           | LINK
```

A record is a list of fields, each having a name (selector) and a specification of the type of values the field can hold. Such values can be either simple (INT, REAL, BOOL, CHAR (n)) or pointers to the record class being defined (LINK's). To illustrate, consider the following example:

```
TYPE person = RECORD (name: CHAR (12) ;
                      age, income: INT ;
                      father, mother: LINK) T_END
```

TYPE and T_END are delimiters for type definitions. Type definitions were introduced in Pascal and are a feature of SPL. In the above a record class called "person" is defined, having " father" and "mother" as selectors for fields of the type LINK (in other words, "father" and "mother" are fields that can contain as value addresses of instances of the record type "person").

Pointer types are defined as follows.

```
pointer-type → POINTER (record-class-identifier)
record-class-identifier → identifier
```

Record class identifiers are names which are assigned to record types through a type definition ("person" in the example above). Note that in this way pointers are syntactically bound to record classes.

As an example, consider the pointer declaration below:

```
DECLARE P1, P2, P3: POINTER (person) D_END
```

P1, P2 and P3 are assigned to memory positions which can hold addresses of instances of the record type "person".

3.3. THE MANIPULATION OF RECORDS AND POINTERS

A record definition does not cause memory to be allocated. This is done by the programmer, through the CREATE statement. Suppose A1 is the name of a variable declared of a certain record type R, and that P1 is the name of a pointer to instances of the same record R. Then the statement

```
CREATE A1 SET(P1) ;
```

causes the allocation of an area in memory compatible with the description of the record class R and sets P1 to point to this area. The general form of a CREATE statement is as follows:

```
CREATE record-identifier SET (pointer-identifier-list);
```

where record-identifier is the name of a variable of type record, and pointer-identifier-list is a list of names of pointers to that same record.

Record instances can be deleted explicitly by the programmer. The execution of a statement of the form

```
FREE  record-identifier  REF(pointer-identifier-list);
```

causes the memory positions occupied by the instances of the record identifier pointed at by the pointer identifiers in the list to be returned to the list of available space. After this takes place, all pointers in the list have their values set to NIL.

Pointer assignment statements are allowed in SPL. Their form is a follows:

```
pointer-variable := pointer-value;
```

Pointer variables are defined as follows:

```
pointer-variable  →  simple-pointer-variable
                     {→record-identifier . link-identifier}*

simple-pointer-variable  →  simple-variable

simple-variable  →  identifier [(subscript-list)]
```

Pointer variables can be either simple or qualified. Simple pointer variables are either identifiers (declared of type POINTER), components of arrays of pointers or function calls returning a pointer value. Qualified pointer variables allow programmers to access instances of records through the link type fields of records in the class.

Pointer values are either the special value NIL or a pointer variable.

```
pointer-value  →  NIL
               |  pointer-variable
```

Pointers can help access record fields of types other than LINK, whose values can be used in expressions. The only other operations allowed on pointers are the comparison operations (= or ¬ =).

3.4 IMPLEMENTATION MODEL

In this section we will show how pointers and links are implemented in SPL. We begin this description by giving an overview of the symbol table organization used in our implementation. Then, all data structures used in connection with pointers are presented, each followed by a brief description of its use. To close this section, we explain how we manage the symbol table structures, with respect to pointer, link and record implementations.

3.4.1 General Description of the Symbol Table Structure

In the SPL implementation, the symbol table has been structured as a balanced tree [KNU 68]. Each node of the tree has the following format:

TREE_NODE:

LINK_GT
LINK_LT
INST_LIST
VALUE

Where the fields have the following meanings:

LINK_GT	is a link to the node to the right, which corresponds to an identifier of higher (alphabetical) order; if such an identifier does not exist in the program, this field has the value NIL.
LINK_LT	is a link to the node to the left, which corresponds to an identifier of lower (alphabetical) order; if such an identifier does not exist in the program, this field has the value NIL.

INST_LIST	points to a list of instances of the identifier.
VALUE	contains the value of the identifier.

The tree is artificially balanced by having as a root a node corresponding to a dummy identifier, whose characters are taken from a central part of the alphabet. The purpose of such a tree structure is to provide quick access to any node, while allowing an efficient sorting of the identifiers in any order. As long as the number of identifiers in a program (excluding reserved keywords) does not become very large, this type of organization seems to be a good choice, and it was tested in practice with good results.

Each identifier in a program may have one or more instances, created by the appearence of the identifier in a prologue. The instances of a given identifier are collected in a list, called an INST_LIST . Each node of this list (called an INST_NODE) has the format below.

INST_NODE:

LINK_INST
TREE_NODE_PT
XREF_LIST
BLOCK_NUMBER
DESCRIPTOR_NODE_INDEX

Where

LINK_INST	is either a link to the next INST_NODE, or it has the value NIL.
TREE_NODE_PT	is a pointer back to the TREE_NODE.

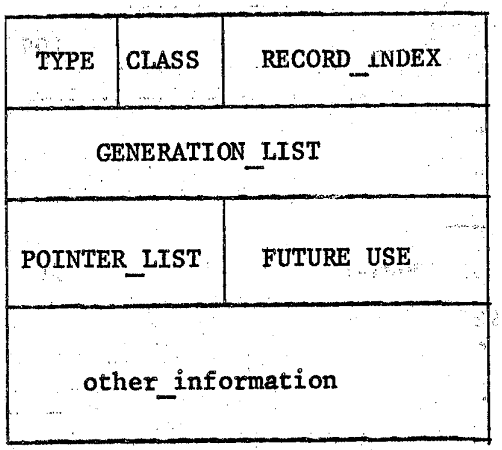
XREF_LIST	is a pointer to a list of references (this list is maintained for the purpose of emitting a cross reference list at the end of the compilation).
BLOCK_NUMBER	is the number of the block in which this instance was declared.
DESCRIPTOR_NODE_INDEX	is an index to a node describing the type of declaration (procedure, record, pointer, etc.).

The format of the descriptor node varies with the particular type being described. We will be concerned only with descriptors for pointer types. These will be shown in the next subsection.

3.4.2. Data Structures Used to Implement Pointers

Each pointer variable, or link field, has an associated descriptor node with the following format:

DESCRIPTOR_NODE:



Where:

TYPE	gives the type of the descriptor node. We will write TYPE='P' for a pointer type and TYPE='L' for a link type.

CLASS gives the class of the reference, i.e, if it is a parameter, a common variable, a type definition, or a record field.

RECORD_INDEX is an index to an entry in the symbol table containing the INST_NODE of the record pointed at by this pointer; this index is used to check whether in any reference involving this pointer, it points to the correct record, or record class.

GENERATION_LIST is a doubly linked list of all pointers and links that point to a given generation of a record; thus, if a record generation is destroyed (freed), all pointers in this list can be set to logically point to NIL. Actually, this field is composed of two pointers: a FORWARD_PT that points to the next item, and a BACKWARD_PT, that points to the previous one.

POINTER_LIST is a list of all pointers and links declared in a given block; when the end of the block is reached, this list is scanned and all pointers in it are removed from their GENERATION_LIST's (if any), thus logically pointing to NIL.

other_information is the global name given to the remaining of the node. It will probably contain any other information needed by the code generation routines, like internal name, addressing scheme, data length, and so on.

There is one GENERATION_LIST for each generation of a given record. The list is used to collect all pointers that point to this generation. Each GENERATION_LIST has also a GENERATION_LIST_HEAD, which is a node allocated outside the symbol table area when a generation is created. The header has the format:

GENERATION_LIST_HEAD:

LAST_IN_LIST	FIRST_IN_LIST

When LAST_IN_LIST and FIRST_IN_LIST are both null, this means that the list is empty, that is, no pointer or link points to this generation. In this case an error message is issued, telling that some generation has lost all its pointers. It is important to notice that the development of an isolated ring of generations cannot be detected. An isolated ring occurs when various generations of some record point to each other, but no external pointer points to them: the ring has no accessing path. The prevention of such a situation could be done only at run time, using a garbage collector. In our implementation it is programmer's responsibility that such a situation never occurs.

Whenever a pointer or link no longer points to a data structure (which may occur by assignment, block exit, record creation or deletion), it is deleted from its generation list. This action can also cause the insertion of this pointer in another generation list.

3.4.3. Pointer Management

Here we will explain how the data structures previously described are managed in the SPL implementation. We will consider all events that may affect those structures, such as record creation and deletion, pointer assignment, block exit, etc. At the end we give a summary of the error conditions that are effectively detected.

a) Pointer and Link Declarations.

Whenever a pointer or link is declared in a prologue, a descriptor node of the type described in 3.4.2. is created for it. The initial values of the fields are:

- TYPE = 'P' , or TYPE = 'L' if it is a pointer or link, respectively;
- CLASS = 'P' if it is a parameter, or CLASS = 'T' if it is a type definition, or CLASS = 'C' if it is a common variable, or CLASS = 'F' if it is a record field (LINK only).
- RECORD_INDEX points to the instance of the record being pointed at. If it is a pointer declaration, this record must be explicitly declared. In the case of a LINK field, the record is implicitly declared to be the record containing the field. Notice that this record can be both a record variable, or a record class.
- GENERATION_LIST is initially set to logically point to NIL. This is done by making BACKWARD_PT and FORWARD_PT point to the descriptor node itself. So all pointers and links are initialized to NIL.
- POINTER_LIST is linked with the last pointer or link declared in the same prologue, if any.

b) BLOCK EXIT

When the end of a block is reached, the POINTER_LIST is scanned, and for every node in it, the following actions are carried out:

- a consistency check is done, by examining the TYPE field of the node: if it is not a pointer or link, a compiler error has occurred;
- the node is deleted from its generation list, if present in such a list;
- the GENERATION_LIST_HEAD of the generation list from which the pointer has just been deleted is examined to see if the freeing of the pointer does not cause any record instance to lose all its pointers. Notice that this will not be done if the record was declared in the block being exited (in this case the header is also deallocated).

Note that for blocks that do not declare pointers, only a very small amount of overhead is added.

c) CREATE statement

If a CREATE statement of the form

CREATE record-identifier SET (pointer-identifier-list);

is found, a GENERATION_LIST_HEAD is allocated, and for every pointer in the SET list the following actions are performed:

- the RECORD_INDEX field is examined to see whether it indicates the record class of the previously given record_identifier. If a mismatch is found, this particular pointer variable is ignored;

- the pointer is freed from its previous value, if any, since its value was changed by the CREATE statement;

- the pointer is inserted in the generation list being created.

If any garbage collection is performed at run time, it is possible to find any isolated ring that may have been developed during program execution.

d) FREE Statement

When a FREE statement of the form

FREE record-identifier REF (pointer-identifier-list);

is found, each pointer identifier in the REF list is removed from its generation list (if a pointer has the value NIL, an error message is issued). All such pointers must point to instances of record_identifier, or to instances of its record class: if this is not the case, an error message is issued. If the record being freed contains any LINK fields, these will also be removed from their generation lists, if any. Before deleting a pointer from its generation list, the list is scanned and all other pointers (and links) still pointing to this generation are set to point to NIL. The GENERATION_LIST_HEAD's encountered are deallocated.

e) Use of Pointers

In a qualified reference of the form

simple-pointer-variable $\{\rightarrow$record-identifier.link-identifier$\}^*$
$\rightarrow$record-identifier.field-name

every pointer or link identifier must point to instances (generations) of the same record_class. This will be tested by comparing the RECORD_INDEX field of the pointer or link descriptor with the index of the INST_NODE of the record class associated with the record_identifier. If a match cannot be found, the reference is invalid. It is at this time that null pointers can also be discovered.

f) Pointer Assignment Statement

A pointer or link can be assigned to another pointer or link. During the assignment statement reduction, the RECORD_INDEX fields of the pointers are compared to see if they point to compatible records. The only exception is when the special value NIL is the source of the assignment: then no validation of the assignment is needed.

Before receiving its new value, the target pointer (or link) must be deleted from its generation list; if this causes the list to be empty, an error message is issued telling the programmer that a generation has lost all its pointers.

The final step consists of adding the target pointer or link descriptor to the source's generation list.

g) Summary of Error Conditions

The following error conditions can be detected in the previously described implementation model:

- invalid pointer, or link qualification;
- loss of all pointers to a given generation of a record.

The following error conditions can never occur, because of the syntax definition:

- dangling references, because a pointer cannot be declared in a scope wider than that of the record it can point at;
- if a block exit frees all pointers and links that may point to a given record, it is impossible, by the scope rules, to refer again to this record. However, such a situation causes the appearance of a warning message, which is done for completeness purposes only.

4. CONCLUSIONS

SPL is a programming system designed for beginners in Computer Science. As such, some of its features were given special attention. In particular, language structures which in other systems may give rise to undisciplined programming were carefully adapted to meet the purposes of SPL. In this paper we showed how pointers and records are defined and implemented in the SPL system. We stated that the only reason for the presence of pointers and records in our language is to give programmers the ability to model and manipulate list structures in a more natural way. Some well known problems associated with the use of pointers in programming languages (type violations and dangling references) were avoided in our system.

APPENDIX: The syntax of SPL

Note: nonterminals not defined here are assumed to be as in Pascal.

```
program → block

block → BEGIN prologue statement-list END

prologue → [TYPE type-definition-list   T_END]
            DELCARE declaration-list     D_END

type-definition-list → type-definition {; type-definition }*

type-definition → type-identifier = type

type-identifier → identifier

declaration-list → variable-declaration-list [routine-declaration-list]
                 | routine-declaration-list

variable-declaration-list → identifier-list: type {; identifier-list:type}*

identifier-list → identifier {, identifier }*

type → basic-type
     | structured-type
     | type-identifier

basic-type → simple-type
           | pointer-type

simple-type → INT | REAL | BOOL | CHAR (unsigned-integer)

pointer-type → POINTER (record-class-identifier)
```

```
record-class-identifier  →  identifier

structured-type  →   array-type

                 |   record-type

array-type  →  ARRAY (bound-pair-list) OF basic-type

bound-pair-list  →  bound-pair  {, bound-pair }*

bound-pair  →  [ simple-expression : ] simple-expression

record-type  → RECORD (field-list)

field-list  → field-name {, field-name }* : field-type
             { ; field-name {, field-name }* : field-type }*

field-name  →  identifier

field-type  →  simple-type

            |  LINK

routine-declaration-list  → routine-declaration
                            {routine-declaration}*

routine-declaration  →  procedure-declaration

                     |  function-declaration

procedure-declaration  →  PROCEDURE  procedure-head  routine-body   P_END

function-declaration   →  FUNCTION  function-head   routine-body    P_END

procedure-head  →  procedure-identifier [(procedure-formal-parameters)]

procedure-identifier  →  identifier
```

```
procedure-formal-parameters → procedure-access-mode
                              formal-parameter-declaration
                              {; procedure-access-mode
                                 formal-parameter-declaration }*

procedure-access-mode → INPUT | REFER | OUTPUT

formal-parameter-declaration → formal-parameter-list:formal-type

formal-parameter-list → identifier-list

formal-type → basic-type

            | array-type

routine-body → [prologue] statement-list

function-head → function-identifier [(function-formal-parameters)]

                RETURNS  basic-type

function-identifier → identifier

function-formal-paramters → function-access-mode
                            formal-paramter-declaration
                          {;function-access-mode formal-parameter-declaration }*

function-access-mode → INPUT | REFER

statement-list → statement { statement }*

statement → simple-statement

          | structured-statement
```

```
simple-statement → assignment-statement
                 | exit-statement
                 | i/o-statement
                 | create-statement
                 | free-statement
                 | procedure-call-statement
                 | return-statement

assignment-statement → variable := expression;
                     | pointer-variable := pointer-value;

expression → [ expression  OR ] logical-product

logical-product → [ logical-product  AND ] relation

relation  → [ simple-expression  relational-operator ]
              simple-expression

simple-expression → [ simple-expression addition-operator ] term

term → [ term multiplying-operator ] factor

factor → memory-reference
       | addition-operator  factor
       | NOT  factor
       | (expression)

memory-reference  → constant
                  | variable

constant →  unsigned-number
         |  string
         |  boolean-constant
```

```
unsigned-number → unsigned-integer
                | unsigned-real

unsigned-integer →  digit { digit }*

unsigned-real  → unsigned-integer . unsigned-integer
                 [  E  addition-operator  unsigned-integer ]

variable  → simple-variable
          | pointer-variable → record-identifier. field-name

simple-variable → identifier [(argument-list)]

pointer-variable → simple-pointer-variable
                   { →record-identifier.link-identifier }*

simple-pointer-variable → simple-variable

pointer-value → NIL
              | pointer-variable

exit-statement → EXIT ;

i/o-statement  → READ (i/o-list);
               | READON (i/o-list);
               | WRITE (i/o-list);
               | WRITEON (i/o-list);

i/o-list → identifier-list

create-statement → CREATE record-identifier SET
                   (pointer-identifier-list);

record-identifier → identifier
```

```
pointer-identifier-list → identifier-list

free-statement → FREE record-identifier REF
                 (pointer-identifier-list);

procedure-call-statement → CALL procedure-identifier
                           [(argument-list)];

argument-list → memory-reference {,memory-reference}*

return-statement → RETURN [expression ];

structured-statement → compound-statement
                     | conditional-statement
                     | repetitive-statement

compound-statement → block
                   | group-statement

group-statement → GROUP statement-list  G_END

conditional-statement → if-statement
                      | case-statement

if-statement → IF expression THEN statement [ELSE statement]  I_END

case-statement → CASE simple-expression OF case-list  C_END

case-list → case-element  {; case-element }*

case-element → unsigned-integer {, unsigned-integer}* : statement

repetitive-statement → while-statement
                     | repeat-statement
                     | for-statement

while-statement → WHILE expression DO statement  W_END

repeat-statement → REPEAT statement UNTIL expression  R_END

for-statement → FOR identifier : = for-list DO statement  F_END

for-list → simple-expression TO simple-expression
           [ STEP simple-expression ]
```

REFERENCES

BER 70 BERRY, D.M. Block structure: retention or deletion? Providence, Brown Univ., Center for Computer and Information Sciences, 1970. TR-29.

BER 75 BERRY, D.M. Correctness of data representations: pointers. Draft report.

CHI 73 CHIRICA,L.M. et alii. Two EULER run time models: the dangling reference, imposter environment and label problems. In: ACM/IEEE SYMPOSIUM ON HIGH LEVEL LANGUAGE COMPUTER ARCHITECTURE, College Park, Md, Nov. 73. p. 141-51.

HOA 72 HOARE, C.A.R. Notes on data structuring. In: DAHL, O.J. et alii. Structured programming. London, Academic Press, 1974. p.83-174.

HOA 73a HOARE, C.A.R. Hints on programming language design. Stanford, Stanford Univ., Computer Science Department, 1973. CS-403.

HOA 73b HOARE, C.A.R. Recursive data structures. Stanford, Stanford Univ., Computer Science Department, 1973. CS-400.

HOA 75 HOARE, C.A.R. Data reliability. SIGPLAN Notices, 10 (6): 528-33, June 1975.

HOL 72 HOLT, R.C. Teaching the fatal disease. Toronto, Univ. of Toronto, Department of Computer Science, 1972. RCH-1.

KNU 68 KNUTH, D.E. Fundamental algorithms. In: ______. The art of computer programming. Reading, Addison-Wesley, 1968. V.1.

LZ 74 LISKOV, B. & ZILLES, S. Programming with abstract data types. SIGPLAN Notices, 9 (4) : 50-9, Apr. 1974.

vWn 69 VAN WIJNGAARDEN, A. et alii. Report on the algorithmic language Algol 68. Amsterdam, Mathematical Center, 1969.

WH 66 WIRTH, N. & HOARE, C.A.R. A contribution to the development of Algol. Commun. ACM , 9 (6) : 413-32, June 1966.

WIR 71a WIRTH, N. Program development by stepwise refinement. Commun. ACM , 14 (4) : 221-7, Apr. 1971.

WIR 71b WIRTH, N. The programming language Pascal. Acta Informatica, 1 : 35-63, 1971.

ZEL 74 ZELKOWITZ, M.V. Pointer variables within a diagnostic compiler. College Park, Univ. of Maryland, Department of Computer Science, 1974. Tr-343.