PUC

Series: Monographs in Computer Science
and Computer Applications
Nº 6/75

PEP: A LANGUAGE FOR PROVABILITY, EFFICIENCY AND PORTABILITY

by

Sergio Carvalho
Carlos J. Lucena
Daniel Schwabe
P.L. Rosa Filho

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 6/75

PEP: A LANGUAGE FOR PROVABILITY, EFFICIENCY AND PORTABILITY *

by

Sergio Carvalho
Carlos J. Lucena
Daniel Schwabe
P.L. Rosa Filho

Series Editor: Larry Kerschberg

October, 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:

PEP is a very high level language aimed at Provability , Efficiency and Portability. The language PEP has a comprehensive set of data and control structures, all of which can be easily defined axiomatically. PEP user's have the ability to define abstract data types. The language is informally described, and an implementation model is presented.

KEY WORDS:

Very high level language, provability, efficiency, portability, control structures, abstract data types, implementation model, data structures.

RESUMO:

PEP é uma linguagem de muito alto nível que visa a Provabilidade, Eficiência e Portabilidade . A linguagem PEP possui um amplo conjunto de estruturas de controle e de dados que podem ser facilmente definidos de forma axiomática. Os usuários da PEP podem definir tipos de dados abstratos. A linguagem é descrita informalmente, apresentando-se um modelo de implementação.

PALAVRAS CHAVE:

Linguagem de muito alto nível, provabilidade, eficiência, portabilidade, estruturas de controle, tipos de dados abstratos, modelo de implementação, estruturas de dados.

CONTENTS

1. INTRODUCTION

This paper gives an overview of the very high level language PEP. This language has been designed* with the objective of experimenting with the idea of using a number of issues from software engineering as a metric for language design.

The authors have attempted to bridge the gap between program specification and program implementation by proposing a language of very high level (very general models for data abstrations). A program encoded in PEP does not refer to implementation level data structures. This fact plus the ease of verification of PEP's control structures assure that programs can be easily proven (or checked for consistency) at the specification level. A specification in PEP (a PEP standard program) can be automatically compiled to a code which will be executed following a simple implementation model.

An alternative approach to code generation from a specification in PEP consists of replacing the direct compilation by the manual production of multi-level programs (at a lower source language level) that are said to implement PEP's data types. Following this approach it is possible either to propose a particular implementation for PEP's standard structured types (which are of a very high level) or to propose implementations for user's defined abstract data types (not included in the language's definition).

The approach is aimed at allowing PEP's compiler to generate code for data structures specified in more detail, thus enhancing more efficiency of the object code. The property of portability is achieved because of the way the multi-level implementation is set-up.

---

* Some aspects of the language are currently being implemented.

In the next sections we describe the general structure of the language, define its standard data types and present its implementation model. Finally we discuss how abstract data types and alternative implementations of standard data types can be accomplished.

2. STRUCTURE OF THE LANGUAGE

In this section the general structure of the language PEP is informally described. To do so we use an extended BNF notation, where:

- words in non-capital letters denote nonterminals;
- words in capital letters denote reserved identifiers in the language;
- the symbols → , [,] , + , * , | are used as meta-symbols in the language description, with the usual meaning.

A program in PEP consists in a prologue followed by a sequence of modules. Comments (strings of symbols enclosed by " and ") can appear anywhere a blank is permitted.

program → prologue [module]+

The prologue is used to define global information (which can be accessed from each and every module of the program). Both type definitions (as in Pascal [Wir 72]) and declarations of identifiers for variables and routines may be present in the program global area. The prologue is algo used to name the main module of the program (the one which should be executed first) and the list of module names of the program.

```
prologue → GLOBAL  [ type-definition ]*
                   [ declaration ]*
                     MAIN module-name IN module-name-list
           G-END
```

module-name-list → module-name [module-name]*

module-name → identifier

Modules are program units which can be compiled separately, provided the prologue information (if any) is made available to the compiler. This is so because there is no type specification for identifiers in the global access specification, as we shall see later. All modules possess a distinct module name, an optional module head, and a module body.

module → MODULE module-name:
module-head
module-body
M-END

The module head is a context specification area. In it we specify:

(i) how the module communicates with the global area established by the prologue;

(ii) the sequence of type definitions (if any) to be known throughout the module;

(iii) the sequence of identifier declarations local to the module.

It is important to note that modules can communicate only with the global (prologue created) area, and are not allowed to share among themselves type definitions and declarations contaimed in module heads. If such need arises, the information to be shared must he inserted in the prologue created area, and proper mention to that information must be made in the external section of the corresponding module heads.

module-head → [external-part]
[type-definition]
[declaration]

In the external part those types and identifiers created in the prologue, which the module must use, are named. Access can be made by:

(i) READONLY - The module can extract information from the global location, but is not allowed to modify it;

(ii) READWRITE- The module can both extract and modify information contained in the global area.

It should be noted that the names of all global types, global routines and modules accessed by the module must appear in the READONLY list.

```
external-part → READONLY identifier-list [READWRITE identifier-list;]
                |READWRITE identifier-list;
identifier-list → identifier [, identifier ]*
```

The type definition and the declaration sections of the module head are used to create types and identifiers local to the module. As mentioned before, types may be defined as in Pascal. All identifiers used in the module body must be declared in the declaration section of the module head.

```
type-definition → TYPE type-identifier = type
                      [; type-identifier= type]*
                  T-END
type-identifier → identifier
declaration → DECLARE decl-list  D-END
decl-list → variable-declaration [ ; routine-declaration ]
          | routine-declaration
variable-declaration → identifier-list : type [ INIT type-value]
                       [;identifier-list: type [ INIT type-value]]*
```

Types in PEP are defined in sections 4 and 5 below. The INIT option can be used to assign initial value to the declared variables. Note that type and type-value must correspond (no type conversions are performed).

Routines in PEP are of two kinds:

(i) PROCEDURES- do not return values, affecting the computation only through side effects;

(ii) FUNCTIONS - must return a value of a specified type, and are not allowed to have side effects.

The characteristics above are implemented through the proper use of parameter passing and global accessing mechanisms, as is done in STRUGGLE [BER 74].

```
routine-declaration → [ procedure-decl | function-decl ]+
procedure-decl → PROCEDURE
                     procedure-head
                     routine-body
                 P-END
function-decl → FUNCTION
                     function-head
                     routine-body
                R-END
```

Procedure and function heads contain in general:

(i) the name of the routine;

(ii) a list of formal parameters;

(iii) global access spacification;

(iv) local type definitions and declarations.

In addition, function heads must contain the basic type of the value to be returned.

```
procedure-head → procedure-name:
                 procedure-formal-parameters
                 procedure-global-access-specs
                 routine-locals
procedure-name → identifier
function-head  → function-name: return-type
                 function-formal-parameters
                 function-global-access-specs
                 routine-locals
```

function-name → identifier

Procedure parameters can be of three types:

(i) INPUT - passed by value; the type of a formal parameter is the type of the corresponding actual parameter;

(ii) REFER - passed by reference; the procedure is not allowed to change the information referred to;

(iii) OUTPUT- passed by reference;the procedure is allowed to change the information referred to.

Function parameters can only be of the types INPUT and REFER.

procedure-formal-parameters → [INPUT identifier-list;]
[REFER identifier-list;]
[OUTPUT identifier-list;]

function-formal-parameters → [INPUT identifier-list;]
[REFER identifier-list;]

Routines (no matter how deeply nested in modules) can access information created both in the prologue area and in the containing module area (module-head). This is done by specifying in the routine head the type definitions and identifiers the routine wishes to access. This establishes another difference between procedures and functions: while the former can have both READONLY and READWRITE access, the latter is only allowed READONLY access.

procedure-global-access-specs → external-part

function-global-access-specs → [READONLY identifier-list;]

Note that either OUTPUT parameters or READWRITE access must be specified in a procedure head; otherwise there is on way for the procedure to have side effects.

Functions must return a value of a basic type (to be defined in section 4).

return-type → RETURNS basic-type;

In the local section of routine heads new types and identifiers can be created. In particular, routines may be created to an arbitrary depth.

To find the proper environment for an identifier of a routine being executed, we search, in order, the declaration areas of:

(i) the routine being executed;
(ii) the immediately enclosing routine (if any);
(iii) other enclosing routines (if any);
(iv) the enclosing module;
(v) the prologue.

The routine body contains a sequence of statements, to be shortly defined.

routine-body → [statement]+

This concludes the description of a module head. A module body consists of a sequence of statements.

module-body → [statement]+

Two important characteristics of modules are:

(i) modules are not block structured;
(ii) there are no labels in PEP modules.

Statements can be either simple, structured or the return statement. Simple statements are the assignment statement, routine call statement, the module invocation statement and the record creation statement

statement → simple-statement
| structured-statement
| return-statement

```
simple-statement → assignment-statement
                 | procedure-call-statement
                 | module-invocation-statement
                 | record-creation-statement
```

Assignment statements are defined similarly as in Algol W [WH66], and shall not be specified here. Procedure call statements contain the name of the procedure to be executed and an optional list of actual parameters. This list has the same form as a list of procedure formal parameters, already defined. The correspondence between actual and formal parameters is established through the usual left to right rule, the types of the parameters being naturally considered. Functions are invoked by using function names as primaries in expressions, as in Algol 60 [NAU 63]. It should be noted that routines may call routines known [in the Algol 60 sense] at the point of call; as a consequence, all routines are potentially recursive.

```
procedure-call-statement → CALL procedure-name:
                                    procedure-actual-parameters ;
```

Module invocation statements have the form shown below. Since module names are known throughout the program (being declared in the prologue) modules can be invoked from within modules. In particular, modules are potentially recursive.

```
module-invocation-statement → EXECUTE  module-name ;
```

Record creation statements are used to create new instances of record variables, and will be defined in section 4.

Structured statements can be either the conditional statement, the iterative statement or the compound statement.

```
structured-statement → conditional-statement
                     | iterative-statement
                     | compound-statement
```

There is only one conditional statement in PEP. This statement was found to be able to reasonably represent most usual if-then-else

and case constructs which can be generated in programming languages today. Its form is as follows:

```
conditional-statement → IF Boolean-expression THEN statement
                          [;Boolean-expression THEN statement]*
                        I-END
```

Its execution can be described as follows. The Boolean expressions are evaluated in a top-down manner, until either one is found to be true or none is true. In the first case the statement corresponding to this Boolean expression is executed, and execution of the program is resumed with the statement immediately following the I-END delimiter (unless the corresponding statement is a return statement, case in which the execution of the routine containing the conditional statement is ended).

In case no Boolean expression in the list is true, the effect of the conditional statement is null (note that expression evaluation in PEP does not cause side effects).

As an example, consider

```
IF
   a > b THEN a:= a - b ;
   a < b THEN b:= b - a ;
   TRUE  THEN z:= a
I-END
```

Iterative statements are the WHILE statement, the FOR statement and the UNTIL statement. In the descriptions that follow, note that there is no need to enclose the sequence of statements to be repeated by BEGIN-END brackets, due to the special delimeters provided in PEP.

```
terative-statement → while-statement
                   | for-statement
                   | until-statement
```

The WHILE statement is copied from Pascal.

```
while-statement → WHILE Boolean-expression DO [statement]+W-END
```

The FOR statement is in PEP more general than in most programming languages, since control variables may range through values which do not necessarily form an arithmetic progression. The sequence of allowed values for the control variable is expressed by a range specification.

```
for-statement → FOR control-variable IN range-spec
                                    DO [ statement ]* F-END
control-variable → identifier
range-spec → type-identifier [: Boolean-expression]
           | integer-subrange [: Boolean-expression]
integer-subrange → initial-value..final-value [ ,step ])
```

A type-identifier in a FOR statement must have been defined earlier in the program, and must identify either a simple type "enumeration" or a simple type "domain" (see section 4). The control variables in FOR statements with type identifiers standing for range specifications must be declared as being of that type. In the second kind of range specification, initial value, final value and step are integer expressions which define the control variable sequence of values in the usual fashion (in case a step expression is not present, the default value 1 is assumed in its place). In this case the control variable must be declared of type integer. The integer expressions in an integer subrange are evaluated exactly once, before the iteration starts.

Note that since control variables must in either case be defined, their scope is not limited to the FOR statement. This implies that the value of the control variable is available after termination of the loop.

The optional Boolean expression in a range specification can be used to define a condition the next value of the range specification must satisfy in order to be the next value of the control variable.

Examples of FOR statements, where S denotes the statement group to be iterated:

FOR i IN (1..10) DO S F-END

FOR i IN T : rem (i/2) = 0 ∧ i > 10 DO S F-END

Where T is a type identifier previously defined.

UNTIL statements are multiple exit statements. Exit control is implemented through a list of Boolean variables (which must be declared in the corresponding module or routine head). Initially all Boolean (exit) variables have the value "false". Repetition continues until one exit variable is turned to "true" (note that there is no "normal" exit from the loop). When this occurs the repetition is ended and the statement following the UNTIL statement delimeter is executed. Normally this next statement is a conditional statement, where the values of the Boolean exit variables are tested. In the context of an UNTIL statement, the assignment of the "true" value to a Boolean variable in the exit variables list is implicitly made by simply stating the variable name in its desired place. This also signals the closing of the UNTIL statement. Similarly the appearance of a Boolean variable in the exit variables list of an UNTIL statement causes the value of the variable to be set to "false". Finally, it is required that each exit variable in the list must appear in the group of statements to be repeated.

until-statement → UNTIL exit-variable-list DO [statement]$^{+}$U-END

exit-variables-list → Boolean-variable [OR Boolean-variable]$^{+}$

Boolean-variable → identifier

Example of an UNTIL statement:

UNTIL found OR not-found DO S U-END

Compound statements are needed only to group together a sequence of statements to be executed after a Boolean expression in a conditional statement turns out to be "true". Otherwise they are not necessary, due to the special statement delimiters provided in PEP.

compound-statement → BEGIN [statement]$^{+}$ END

Return statements provide one way to end a particular activation of a routine (the other being the normal exit, through the delimiters P-END or R-END).

Note that return statements are semantically meaningful only inside a routine body.

return-statement → RETURN

To conclude this section we present, for illustrative purposes, a skeleton of a PEP program. The program has a global area in which:

- a type T1 is defined;
- variables A, B (of type T1) and C (of type INTEGER) are declared;
- a procedure P is declared, with:
  - parameters X, Y and N;
  - global access to variables A (unmodifiable) and C (modifiable);
  - a group of locally declared identifiers;
  - the body of the procedure.
- the program is said to have two modules, M1 and M2, of which M1 is the leading one.

Module M1 has access to global procedure P, modifiable access to the global variable C, and in it is defined a function F, with formal parameters X and Z, returning a Boolean value. F has (unmodifiable) access to C. After M1's body and the M-END delimiter, module M2 is constructed.

```
GLOBAL
        TYPE T1 = " type specification " T-END
        DECLARE A, B:  T1;
                   C:  INTEGER
                PROCEDURE  P :
                   INPUT  X;
                   REFER  Y;
                   OUTPUT N;
                   READONLY A;
                   READWRITE C;
                   DECLARE " procedure locals "  D-END
                   " procedure body "
                P-END

        D-END
        MAIN M1  IN M1, M2

G-END
MODULE M1 :
    READONLY P;
    READWRITE C;
    DECLARE
          " local variables "
          FUNCTION F:  RETURNS BOOLEAN;
                INPUT X;
                REFER Z;
                READONLY C;
                "function body"
                R-END
    D-END
    "module M1 body"

M-END
MODULE M2:
    "M2 external access"
    "M2 locals"
    "M2 body"
M-END
```

3. STANDARD DATA TYPES

Types in PEP can be standard or abstract.

```
type → standard-type
     | abstract-type
```

In this section we describe the standard data types of PEP. The syntax, semantics and implementation of abstract data types shall be considered in section 5. For a PEP programmer who wants to use an abstract data type, it suffices to know the following:

```
abstract-type → ABSTRACT (basic-type) WITH operations-list
operations-list → operation-name [, operation-name ]*
operation-name → identifier
```

As an example, the type definition below specifies the need for an abstract type STACK of integers, with the operations PUSH, POP, TOP and EMPTY.

```
TYPE STACK = ABSTRACT (INTEGER) WITH  PUSH, POP, TOP, EMPTY T END
```

We shall now consider standard data types.

```
standard-type → simple-type
              | structured-type
              | pointer-type

simple-type   → basic-type
              | enumeration
              | domain

basic-type    → INTEGER
              | REAL
              | BOOLEAN
              | CHARACTER
```

Enumerations in PEP are similar to the scalar types of Pascal, and are as well considered to establish an order among its constants.

```
enumeration → ENUM (enum-constant [ , enum-constant]* )
enum-constant → identifier
```

As an example, consider the enumeration definition below:

```
TYPE div12= ENUM (one, two, three, four, six, twelve) T-END
```

Another simple type available in PEP is the domain. Integer values constructed according to powerful formation rules can be represented by domains.

```
domain → DOMAIN  (domain-former [; domain-former ]*)

domain-former → domain-expression, ident-values-spec
                [ , ident-values-spec ]  [ : condition]

domain-expression → arithmetic-expression
ident-values-spec → identifier  IN  id-values

id-values → type-identifier
          | integer-subrange
```

The following remarks hold concerning the definition above:

(i) all domain expressions in a domain must be of type integer; all identifiers in a domain expression must be of type integer;

(ii) each and every integer variable appearing in a domain expression must have the corresponding set of values stated in an identifier values specification;

(iii) in the trivial case a domain expression is an integer number, there is no corresponding list of identifier values specification, and no condition;

(iv) the possible values for variables in a domain expression can be stated either by a type identifier corresponding to another (previously defined) domain type or by an integer subrange, as defined in section 2.

The values belonging to a domain type are those generated by all domain formers when the integer variables appearing in the corresponding domain expressions take on their corresponding values, provided the conditions (whenever present) are "true". If some condition is not present then all values generated by the corresponding domain expression are in the domain.

The identifiers in the list of identifier values specification in a domain former are assumed to be nested, the rightmost one in the list being the most deeply nested one, and so on. Domain former values are ordered according to this nested priority, as in a nested group of iterations. Domain values are ordered from left to right, obeying each domain former internal order.

Each new value generated is checked against the ones already in the domain; if already present, a value is not duplicated in the domain.

As an example, consider another way of representing the divisors of 12:

```
TYPE div12a = DOMAIN (x, x IN (1..12) : rem (12,x) = 0) T-END
```

The condition in the definition of a domain can be either a Boolean expression or a quantifier expression.

```
condition → Boolean-expression
          | quantifier-expression
```

In a quantifier expression the universal (∀) and existential (∃) quantifiers may be used. The general form of quantifier expressions is:

```
quantifier-expression →  quantifier ident-values-spec
                         [, quantifier ident-values-spec ]*
                         :  Boolean-expression

quantifier →  [ ¬ ] ∃
           |    ∀
```

Note that the identifiers in quantifier expressions are not necessarily those appearing in the domain former expression, as is the case when the condition is a Boolean expression.

The domain below contains all prime numbers between and including 1 and 100

```
TYPE prime = DOMAIN (1 ;

                        x, x IN (2..100) :
                        ¬∃ y IN (2.. x-1) :
                          rem(x,y) = 0)     T-END
```

PEP provides four structured types: arrays, records, sets and tuples:

```
structured-type → array-type
                | record-type
                | set-type
                | tuple-type
```

Array types in PEP are defined below. No dynamic arrays in the sense of Algol 60 are allowed. Memory space for arrays is allocated when array identifiers are declared.

```
array-type  →  ARRAY (bound-pair [, bound-pair ]* ) OF base-type

bound-pair  →  integer-constant.. integer-constant
```

The type of array components(the base type) can be specified by a previously defined type identifier.

```
base-type → standard-type
          | type-identifier
```

Example:

```
TYPE T = ARRAY (-5..5, 0..10) OF div12 T-END
DECLARE A :  T D-END
```

Elements of array A can assume as values the enumeration constants of type div12.

Records in PEP are similar to records in Pascal, with no variant part.

record-type → RECORD (field [; field]*)

field → field-identifier [, field-identifier]* : base-type

field-identifier → identifier

As is the case of arrays, type identifiers appearing as base types must have been defined before.

Record types are used mainly to represent linked lists. Lists can be implemented through the use of another standard(and un structured) type of the language PEP, the pointer type, which we now describe.

The values which can be possessed by pointer variables are addresses of instances of variables declared of type record. More particulary, pointer variables are assigned, in their declaration, to one certain record type, as seen below. Any attempt to set such a variable to any other variable, not of the type record or not of the particular record to which the pointer variable is assigned, causes an error.

pointer-type → POINTER record-type-identifier
record-type-identifier → type-identifier

It should be noted that pointer type fields within a record of type A must refer to instances of the record type A. Otherwise a pointer variable declared to point to A could eventually end up indicating some instance of a different record B.

To illustrate, consider the program section and the remarks below:

```
TYPE   P = POINTER PERSON ;
PERSON = RECORD (name: CHARACTER;
                 age,salary : INTEGER ;
                 marital-status : BOOLEAN;
                 father : P )
T -END
```

```
DECLARE Q, R : P ;
     EMPLOYEE  :  PERSON
D-END
```

Remarks:

- P is the name of a pointer type. Variables declared of type P (Q, R) can only point to instances of variables declared of the (record) type PERSON;
- PERSON is the name of a record type (otherwise a compile time error would occur) containing five fields: "name", "age", "salary", "marital-status" and "father". All fields are of basic types, except the last one, declared of type P (which is a pointer to PERSON) ;
- Variables Q and R are pointers to instances of "PERSON";
- Variable EMPLOYEE is a record of type PERSON.

Contrary to arrays, record variables are not created when declarations are processed, at compile time. Instead, they are created dinamically, as we shall see below. When record types are defined they merely set a pattern which variables declared of that type must obey. New instances of record variables are created by the record creation statement below:

```
record-creation-statement → NEW  record-variable SET pointer-variable;
record-variable  →  identifier
pointer-variable →  identifier
```

Execution of this statement creates a new instance of a variable declared of a certain record type, and sets to this instance an allowed pointer.

Example: NEW EMPLOYEE SET Q ;

Instances of record variables are automatically deleted when are not being pointed at by any variable.

To conclude the presentation of record types, we remark:

- fields within records can be accessed through their names, qualified by an allowed pointer, as for instance

```
Q. age      (an integer value)
Q. father   (a pointer value )
Q. father.age  (an integer value)
```

- pointer assignment statements are allowed in PEP. So

```
Q := R ;        R := Q. father ;
```

are valid assignment statements since both Q, R and father are declared as pointers to the same record class.

Another structured type in PEP is the set. All elements of a set in PEP must be of the same base type; otherwise sets in PEP are similar to sets in SETL [SCH 73].

set-type → SET OF base-type

Example:

```
TYPE S = SET OF INTEGER T-END
DECLARE  A : S  D-END
```

Given the above the set {1,5, 11, 17 } is an allowed value for variable A.

Sets can be assigned to variables in two ways in PEP:

(i) by explicit assignment as in

```
A:= {1,5,11,17};
```

(ii) by iteration. In this case a set former (similar to the domain formers already seen) is the right hand side of a set assignment statement.

Example:

```
DECLARE   S :  SET OF INTEGER  D-END
```

S:={ p, p IN (2..100) : ¬∃ y IN (2..p-1): REM (p,y)=0};

The value of variable S is the set of all prime numbers (integers) between 2 and 100.

It is important to note that elements of a set are not assumed to be ordered.

The following operations and tests are allowed on set variables (say A, B) and set elements (say x) :

- membership test : x ∈ A ;
- equality test : A = B;
- inclusion test: A ⊂ B;
- set cardinality : CARD (A) ;
- powerset : POW (A);
- set union : A ∪ B ;
- set intersection : A ∩ B ;
- set difference : A - B.

To conclude this section on standard data types, we present the structured type tuple. Tuples in PEP are based on tuples in SETL. They differ from sets in that tuples are ordered sequences of elements; they differ from tuples in SETL in that all elements of a tuple in PEP must be of the same base type:

tuple-type → TUPLE OF base-type

Tuples can be assigned to tuple variables explicitly, as follows:

```
DECLARE  A,B :  TUPLE OF REAL ;
                C :  REAL  D-END

  A:= < 1.0, 2.1, 3.1 > ;
  B:= < 5.0 > ;
```

Tuples can be manipulated in several different ways.

A tuple can be appended to another tuple through the concatenation operator || , as for instance in the assignment statement below:

A:= A || B ;

yielding as the new value for A the tuple < 1.0, 2.1, 3.1, 5.0 >.

The fact that tuples are ordered allows for the use of operations of indexing, extracting and repetition . The ith element of a tuple A can be accessed directly by means of the notation A [i]. In the example below the real variable C is assigned the (real) value of the second element of the tuple A :

C:= A [2] ;

Sections can be extracted from tuples and assigned to other tuple variables, as in the example below

B:= A [1:2] ;

The first element in the bracketed pair indicates the beginning of the desided section; the second element indicates the length of the section . After the assignment above, B has the value <1.0, 2.1>. The expression B * 3 has the value

< 1.0, 2.1, 1.0, 2.1, 1.0, 2.1 >.

Other operations and tests in tuples:

- equality test : A = B ;
- membership test : C ∈ A ;
- cardinality : CARD (A) ;
- head, tail, initial, last :

HEAD (A) $\overset{\text{def}}{=}$ A [1 : 1]
TAIL (A) $\overset{\text{def}}{=}$ A [2 : CARD (A)]
INITIAL (A) $\overset{\text{def}}{=}$ A [1: CARD(A)-1]
LAST (A) $\overset{\text{def}}{=}$ A [CARD(A) : 1]

4. THE IMPLEMENTATION MODEL

Basically a stack model is adequate to support an implementation of the language PEP, due to the following language characteristics:

(i) all modules and routines are potentially recursive;

(ii) there is no storage retention: allocations and deallocations are processed on a first in last out basis.

However, a heap is also needet to implement records, sets and tuples, which can grow dynamically.

Execution of a PEP program consists of a sequence of snapshots, each of which having two components: an algorithm and a record of execution. The algorithm is some representation of the given program, and is invariant throughout the computation. A possible representation is a nested set of contours, as in [JOH 71].

The record of execution consists of a processor, a stack and a heap. The stack contains a series of activation records (AR'S), one on top of the other. There are three kinds of AR'S in PEP, corresponding to the prologue, modules and routines.

a. Prologue AR

Its format is indicated in Fig. 1 below.

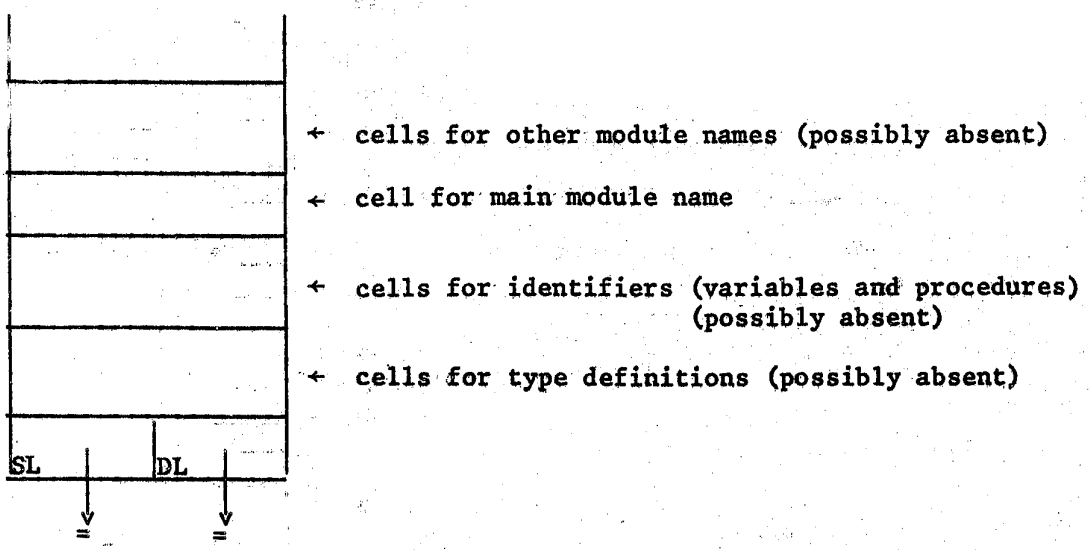


Figure 1. Prologue AR

Both the static link (SL) and the dynamic link (DL) of the prologue AR have the value nil. The prologue AR is generated at the beginning of the computation, when the program prologue is being processed. It sits on the bottom of the stack for the duration of the computation, getting deallocated upon program termination.

Corresponding cells are generated whenever type definitions are present in the prologue. The formats of such cells vary with the type being defined. Without going into too much detail, it is interesting to note that:

(i) enumeration and domain values can be computed at compile time and can be stored in the stack, in the cells immediately on top of the cell containing the name and type code of the type being defined. Also, this cell needs a pointer to the last (topmost)value cell.

(ii) arrays are also static. Whenever an array type is defined, the corresponding cell in the stack is a dope vector containing information such as number of dimensions, bounds, and the like.

(iii) record type cells contain information such as field names and field types.

(iv) pointer type cells contain a pointer to the corresponding record type.

(v) set and tuple type cells contain the name of the component type.

Global identifiers (variables and procedures) also generate cells in the prologue AR when present in the prologue. These cells have the usual format, unless the variable type has been defined before; in this case, the cell contains a pointer to the corresponding type cell.

Each module in the program generates a cell in the prologue AR, pointing to the first instruction of the corresponding module.

b. Module AR

These are generated when either:

(i) an EXECUTE statement is found in the module being processed;

or (ii) a reserved word MODULE is found during sequential program execution.

The general format of module AR's is given in Fig. 2 below.

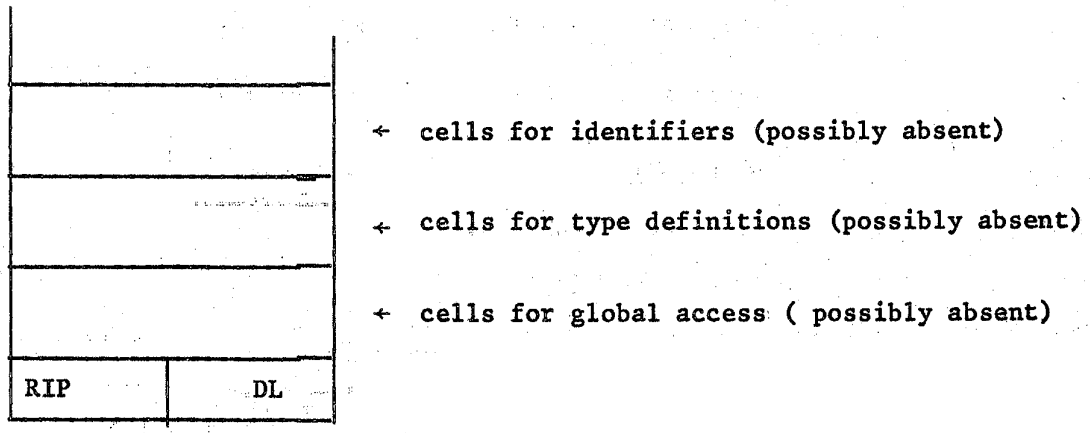


Fig. 2 : Module AR

Module AR's are deleted when the reserved word M-END is found (there is no RETURN from modules).

Due to the scope rules of PEP, the SL of module AR's would, if present, point to the prologue AR. Therefore this information is redundant, and instead of the SL a return instruction pointer (RIP) is used. The RIP points to the instruction to be executed when the module AR in deallocated . Two cases may occur:

(i) Module AR created by EXECUTE statement: the RIP points to the instruction following the EXECUTE statement;

(ii) Sequential entry: RIP points to first instruction of next module.

The DL indicates the base address of the program section being executed when the module AR was created by an EXECUTE statement, or the base

address of the prologue AR if the module AR was created on sequential entry.

Global access cells must contain the access type (READONLY or READWRITE), the name of the (variable, procedure or type) identifier to be accessed, and a pointer to the corresponding global location.

Cells for type definitions and identifiers are as in the prologue AR.

c. Routine AR

Generated when either a procedure is called or a function name is used in an expression. Deleted when either a RETURN statement is executed in the routine body or upon normal routine exit.

Routine AR's are illustrated in Fig. 3 below.

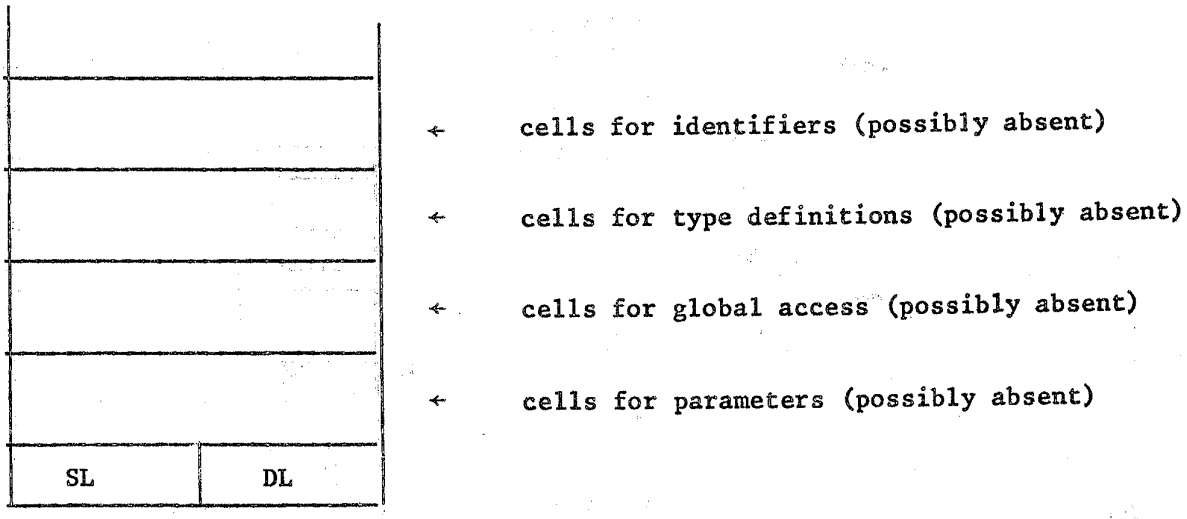


Fig. 3 : Routine AR

The SL points to the bottommost stack position of the AR corresponding to the program section in which the just called routine is declared. The DL points to the bottommost position of the calling program section AR.

Parameter cells must have a field in which the parameter type (INPUT, REFER or OUTPUT) is defined. In addition, the following must be observed:

a. INPUT parameters (passed by value):

- if variable of any simple type, the remainder of the formal parameter cell is like the actual parameter cell;

- if array variable, a new group of stack cells is created, similar to the actual parameter group of cells;

- record and pointer variables cannot be passed as INPUT parameters;

- set (tuple) variables as INPUT parameters cause the creation of a new area in the heap where the set (tuple) values are copied. In the formal parameter cell, a pointer to this new area is set.

b. REFER and OUTPUT parameters (passed by reference):

- Cells for formal parameters contain a pointer to the corresponding actual parameter cell.

Both module and routine AR's use, on top of the local identifiers section, a portion of the stack to perform expression evaluation.

The processor contains four pointers:

ip - instruction pointer, points to the next instruction to be executed;

ep - environment pointer, points to the base of the topmost (the active) AR on the stack;

gp - global pointer, indicates the base of the global AR (the one corresponding to the prologue);

tp - top of the stack pointer, points to the first free cell on top of the current extent of the topmost AR.

The ip is reset during the execution of each instruction, to point to the next instruction. The ep points to the bottommost position of the active AR (the one on top of the stack), and, upon deallocation of an AR, points to the position indicated by the DL of the AR being deallocated.

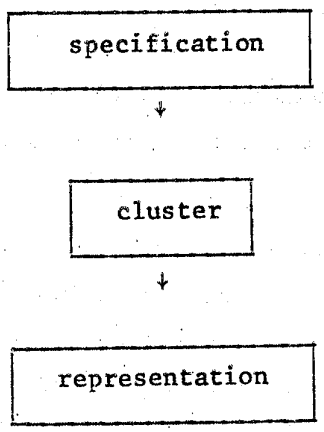
The gp gets set in the beginning of the computation, and never changes. Finally, the tp always points to the top of the stack, being incremented and decremented as expression evaluation proceeds.

The heap is used to store all data structures which can be allocated during program execution (records, sets and tuples). Storage management for sets and tuples is done internally, without the programmer's assistance; however, records and pointers to records are entirely programmer's responsibility.

5. ABSTRACT DATA TYPES AND ALTERNATIVE IMPLEMENTATIONS OF STANDARD DATA TYPES

5.1 Going from Specification to Implementation in PEP.

One of the objectives of PEP is to reduce the gap between the specification of an algorithm for the solution of some problems and its coding into a machine acceplable form. For this, PEP allows for the organization of a program into three distinct levels



The topmost level is called the specification level. In this level the programmer specifies his algorithm using a very high level language (sections 2 and 3). At this level he has available to him the usual scalar data types as well as structured data types.

The second level is the cluster level. Through the use of a modified cluster concept the programmer may use types defined in clusters at this level.

Clusters are also used to re-state the semantics of a possibly general standard type in terms of more basic standard types in the language. In this case, the code produced by the compiler for the data representation and the related operations defined by the cluster will be used in lieu of the code normally produced for the replaced very high level code. In both situations we are interested in the efficiency of the code generated from the cluster.

The cluster approach is due to Liskov and Zilles [LIZ 74] and consists of some language features to model and implement abstract types in terms of operations applicable to objects of the type in such a way that the user of the type needs to be concerned only with the abstract behavior of the type as presented by the operations.

A cluster is an independent external module in which the initialization of the representation of an abstract type T is perfomed, and in which all operations P_j ($1 \leq j \leq n$) related to T are perfomed. The heading of a cluster has the following form:

T : cluster <cluster formal parameters list option>

is P1, P2,..., Pn

The body of the cluster consists of the declaration of the representation (which is visible throughout the cluster) and the code for each P_j .

For more details on clusters, consult [LIZ 74, LIS74].

In a PEP cluster, the programmer may define the abstract type in terms of a flexible representation (third level) and operation bodies written using a standard instruction set for the flexible representation. The standard operations include create, add, sub,select,etc., for creating instances of the data, adding components, subtracting components, etc...

The behavior of the standard instructions on the flexible representation is specified by axioms (these axioms are presented in the appendix).

The flexible representation will usually not be a fixed concrete representation available on the base machine, but rather a higher level representation implementable in several ways.

The actual implementation of the flexible representation is specified in the bottommost level, called the representation level. This level is considered the machine level. Here, the programmer selects a fixed (concrete) representation satisfying the axioms of the flexible representation . Usually (hopefully), these fixed representations will be available from a library; however, they may be coded in yet another cluster. At this level, the standard instruction set is implemented in terms of machine level operations as efficiently as possible.

The use of flexible representations in clusters plus the possibility of having libraries of fixed representations for each base machine will allow us to achieve portability. Performance evaluation of the various commands in the standard instruction set for a set of concrete representations in a particular library will allow us to select a minimum cost fixed representation for a given application [TOM 75,GOT74,LOW74] thus allowing us to achieve the goal of efficiency.

Contrasting a little with sections 2 and 3 we will describe the syntax and semantics of clusters through examples in the following sections we will progressively define our version of cluster and its various uses.

5.2 - Examples of Clusters and Representations

5.2.1 Specification of an abstract data-type through clusters

One of the primary uses of clusters is to create abstract data types at the specification level. As we said before, a cluster is our independent external module. Its definition is the following:

cluster-definition → CLUSTER ON rep-specification WITH operations-list

We will detail this definition through an example. Let us suppose, for instance, that the user needs to use a "stack" in his algorithm. Since there is no "stack" type in the language, he must write a cluster for it. This cluster could look like

```
 1  stack: CLUSTER ON rep 1(INTEGER) WITH push,pop,top;
 2  DECLARE
 3        st : REP;
 4  D-END
 5  PROCEDURE push: INPUT a;
 6     st$add (' - ',a);
 7  P-END
 8  PROCEDURE pop: ;
 9     st$SUB (' _ ');
10  P-END
11  FUNCTION top: RETURNS INTEGER;
12     RETURN st <1> ;
13  R-END
14  END stack
```

Line 1 declares "stack" as being a cluster that will use some representation (physical data structure), that is unknown at this point, which gives a structure to integers; any variable of type "stack" can be operated upon through the operations "push", "pop","top". In line 3, we say that "st" is the name of an instance of the physical data structure that is the representation.

This declaration causes a call to the "create" operation that all representation modules (to be explained later) have; this operation has the effect of allocating space in the memory in a way similar to the one used for variables in the GLOBAL area. The main difference is that this area is accessible only through the standard operations available in the representation module.

That is precisely what the "push" and "pop" operations do; "push" adds a new element to the representation, placing it before the first element, and assigns to it the value contained in "a". Conversely, "pop" subtracts the first element in the representation. Finally, "top" consults this first element; st<1> should be read as st$SELECT (1).

A cluster may also have a "create" operation; in this case, this operation initializes the representation with some values.

In line 1, we notice the keywords repl(INTEGER). This is an instance of a rep-specification. What this actually says is that the representation for this type (cluster) has one (repl) level of structuring; the basic (atomic) type is INTEGER. At this point, we don't specify what concrete representation will this correspond to.

5.2.2 Re-specification of a standard structured type through clusters.

A second use of clusters is to re-specify the semantics of a standard structured data type for reasons of efficiency. The need for this would rise when certain particularities in the use of this type in a program at specification level allow us to simplify some of regular operations defined for that type. Another reason for doing this is when we want to extend or create new operations for that type.

Suppose for instance that we have a set of sets in a specification level program, and due to the logic of the program, whenever a new set is added to this set of sets, it is always disjoint from all the other ones

already in the set. Therefore, the set-add operation would not have to check if the set being added to the set of sets is already there,thus saving execution time.

A cluster for this would look like

```
1  set-of-sets: CLUSTER ON rep2 (rep1 (INTEGER)) with.....,set-add,...;
2     DECLARE   s: REP  D-END
             .
             .
             .
3  PROCEDURE     set-add: INPUT a;
4     DECLARE a rep1(INTEGER) D-END
5     S$ADD('+',a);
6  P-END

      :

7  END  set-of-sets ;
```

In this example,we say in line 1 that the representation for this cluster has two levels of structuring and that the basic type is INTEGER. Clearly, the outermost level (rep2) corresponds to the set (of sets); the innermost level (rep1) corresponds to the sets contained in the set of sets. Since the elements of the outermost set are sets themselves, the parameter for the set-add operation is of the "innermost set" type (line4).

When we have this kind of composite representation (several levels), we must use the "composite" selection. When we write s<i> we select the whole set (rep1(INTEGER)) that is in "position" i in rep2. In the same way, s<i,j> selects the j^{th} element (INTEGER) in the i^{th} set (rep1) in the set of sets (rep2)

As in the first example both rep1 and rep 2 stand for "generic" representations (probably different) that will be associated with a concrete representation module at execution time.

5.2.3 - A representation level program

A representation level program (module) implements a concrete data structure with a standard fixed set of operations. This is the representation module referred to as repi in the previous examples.

Let us see, for example, how a linked list would be implemented:

```
 1  list: REP is RECORD(val: INTEGER; next POINTER REP) ;
 2      DECLARE
 3          r : REP
 4          head, last: POINTER REP ;
 5      D-END
            :
 6  PROCEDURE add: INPUT pos, elem ;
 7          DECLARE
 8            pos : CHARACTER ;
 9          elem:INTEGER
10          link:POINTER REP
11      D-END
12      NEW r SET link ;
13      link.val:= elem;
14      IF
15        pos = '-' THEN BEGIN
16                       link.next:=head;
17                       head.next:=link;
18                       END
19        pos= '+' THEN  BEGIN
20                       link.next := Λ;
21                       last.next := link;
22                       END
23        I-END
24        RETURN
25  P-END
      :
26  END list
```

In line 1 we say that the "template" for the representation is a record with two fields, one for the value ("val") and another for a pointer to the next element("next"). Line 3 says that "r" is of the special type REP, which is actually the template declared in the header. We also have

two auxiliary variables "head" and "last", which will point to the first and last nodes in the list. We assume that these two variables are properly initialized in the "create" operation for this representation. It should be noted that all variables declared in lines 2-5 refer to memory locations that are fixed, i.e., their allocations and de-allocations are controlled explicitly in the operation bodies in the representation, and are not automatically controlled by the module's activation and de-activation. Thus their values remain unchanged between activations of the representation module.

The operation "add" receives two input (by value) parameters, a position ("pos") and a value ("elem") (line 6); it creates a new node with value "elem" (lines 12-13), and links this node in the beginning (lines 15-18) or end (lines 19-22) of the list, according to "pos". Line 20 assigns Λ (null pointer value) to the "next" field of the record pointed to by "link".

5.2.4 - Efficiency Considerations

The efficiency problem can be tackled in many ways in PEP. First, as exemplified before, we can re-specify the semantics of a standard structured data type and increase the efficiency of some operations.

Another way is to gather statistics on the use of standard operations of rep's [TOM75], and also on the use of such operations in a given program. In this way, we would be able to determine which are the crucial operations for that specific program and provide the representation where these particular operations are more efficient.

Note that a rep is supposed to make the best possible use of the resources of a given machine; it does not have to be coded in PEP. Now, supposing that each machine has available a library of representation level programs, with statistics gathered about their operations, it is not difficult to envisage a system where the representations would be selected automatically [LOW74].

Finally, there are some situations when the specification level "drives" the cluster in an inefficient way. For example, consider the statement (b is a sequence)

b:= head (b) || 'a' ||tail (b)

This would be translated into 4 operation calls (head, tail, || , ||). But if we look at this closely, we will notice that this is equivalent to (suppose r is the representation of b)

r $ INSERT ('a', 1, '+') (insert 'a' after the 1st element).

To avoid such situations, the user is able to define "patterns" of operations. Whenever an operation matches the pattern, a specified piece of code is generated, instead of the standard compilation code. This pattern-matching is activated through an option for the compiler.

5.2.5. Portability

Our approach to portability is based on the idea of standardizing the instruction set that operates on the flexible representation. This notion was suggested by the work of Standish [STA73] where an attempt is made to axiomatize the basic properties of all data structures.

Assuming a storage structure that behaves like Standish's data spaces we have converted the selection and assignment operations into a set of convenient basic operations that form the instruction repertoire of our cluster level (i.e., of our flexible representations). These basic operations have a very simple and universal semantics which does not depend on the actual implementation of the flexible representation.

APPENDIX

Let r $\in$ rep(t). Then a value of r is a sequence of t's selected by consecutive integers starting from 1. The length of such a sequence is one less than the first index j such that the selection of the j^{th} element yields Λ. The operations assumed are:

add: rep (t) $\times$ { '+' , '-' } $\times$ t $\rightarrow$ rep (t)

add: (S,p,e) $\triangleq$ add e to the beginning or end (depending on whether p is '+' or '-') of S

sub: rep (t) $\times$ t $\rightarrow$ rep (t)

sub: (S,e) $\triangleq$ remove e from S if e is in S

select: rep (t) $\times$ int $\rightarrow$ t

select: (S,i)=S<i> $\triangleq$ the i^{th} element of S if it exists and Λ otherwise

replace: rep (t) $\times$ int $\times$ t $\rightarrow$ rep (t)

replace: (S,i,e) $\triangleq$ change the i^{th} element of S to e if S<i> $\neq \Lambda$

insert : rep (t) $\times$ int $\times$ {'+' , '-'} $\times$ t $\rightarrow$ rep (t)

insert : (S,i,p,e) $\triangleq$ insert e into S before or after (depending on whether p is '+' or '-') the i^{th} element of S.

In the above, whenever an element is inserted or removed the indices are shifted to preserve the fact that consecutive integers from 1 through the length select non-Λ elements. Formally, we have following axioms:

Let r = rep (t)

1) $v \in r \supset [(\forall i)\ (1 \leq i \leq \text{length } (v) \supset (\text{select } (v,i) \in t \wedge \text{select } (v,i) \neq \Lambda))]$

2) $v = \text{add } (s,'+',e)$ iff [length (v) = length (s) +1 $\wedge$ select (v,1)= e $\wedge$

$((\forall i)\ (1 \leq i \leq \text{length } (s) \supset \text{select } (v,i+1)=\text{select } (s,i)))]$

3) $v = \text{add } (s,'-',e)$ iff [length (v)=length (s)+1 $\wedge$ select (length (v))=e $\wedge$

$((\forall i)\ (1 \leq i \leq \text{length } (s) \supset \text{select } (v,i)=\text{select } (s,i)))]$

4) v=sub (s,e) iff

(∃ j ∋ e = select (s,j) ⊃ [length (v) = length (s-1) ∧

((∀ i) (1≤i≤j-1 ⊃ select (v,i)=select (s,i)))

∧ ((∀ i) (j+1≤i≤ length (s) ⊃ select (v,i-1)=select (s,i)))] ∧

((∄ j) ∋ e = select (s,j)) ⊃ v=s

5) v = s iff (∀ j) (select (v,j) = select (s,j))

6) v = replace (s,i,e) iff

((select (s,i) ≠ Λ) ∧(∀ j)(j ≠ ⊃ select (v,j)=

select (s,j)) ∧ select (v,i)=e)

7) v = insert (s,i, '+', e) ⊃ iff [1≤i≤ length (s) ⊃ (length (v)=length (s)+1

∧ ((∀ j) (1≤j≤i) ⊃ select (v,j)= (s,j))

∧ ((∀ j) (i+1≤j≤ length (s)) ⊃ select (v,j+1)=select (s,j))

∧ select (v,i+1)= e)]

8) v = insert (s,i,'-',e) ⊃ iff [1≤i≤ length (s) ⊃ (length (v) =length(s)+1

∧ ((∀j) (1≤j≤ i-1) ⊃ select (v,j)=select(s,j))

∧ ((∀j) (i≤j≤ length (s))⊃ select (v,j+1)=select (s,j))

∧ select (v,i) = e)]

Notation: v <i>= select (v,i)

REFERENCES

BER 74 BERRY, D.et al. - STRUGGLE - structured generalized goto-less language (preliminary version). Los Angeles, University of California, Computer Science Department, 1974. Memo.-134.

GOT 74 GOTTLIEB, C.C. & TOMPA, F.W. - Choosing a storage schema. Acta Informatica, 2 , 1973.

JOH 71 JOHNSTON, J.B. - The contour model of block structured computations. Sigplan notices, 5 (2), Feb. 1971.

LOW 74 LOW, J.R. - Automatic coding choice of data structures.Stanford, Stanford University, Computer Science Department, 1974. STAN-CS-74.

NAU 63 NAUR, P.ed. - Revised report on the algorithmic language Algol 60. Communications of the ACM, 6 (1) Jan. 1963.

SCH 73 SCHWARTZ, J.T. - On programming: an interim report on the SETL project, Installment I : Generalities. New York, New York University, Courant Institute of Mathematical Sciences, Computer Science Department, 1973.

TOM 75 TOMPA, F.W. - Evaluating the efficiency of storage structures. Waterloo, University of Waterloo, Department of Computer Science, 1975. CS-75-16.

WH 66 WIRTH, N.; HOARE, C.A.R. - A contribution to the development of Algol. Communications of the ACM, 9 (6), June 1966.

WIR 72 WIRTH, N. - The programming language Pascal; revised report. Zurich, Eidg. Technische Hochschule, 1972.