

Khawarizm

Environment of writing abstract algorithms

and of translating them to PASCAL and C languages

Pr D.E ZEGOUR

Ecole Supérieure d'Informatique

S U M M A R Y

1. General presentation

- Presentation
- Menus
- Traitments
- Z Language
- Documentation

2. Steps to follow for the realization of a program under K H A W A R I Z M.

- Familiarization with an arbitrary algorithmic language
- Editing the algorithm
- Syntax check
- Running
- Simulation
- Trace
- Translation to a programming language
- PASCAL or C Programmation
- Example of a Z algorithm
- PASCAL Equivalent
- C Equivalent

3. Z language (Basic)

- Overview
- Structure of a Z algorithm
- Control structures
- Simple actions
- Scalars
- Pointers
- Expressions
- Comments
- High level operations
- Standard functions
- Fonctions de génération aléatoire
- Functions on strings
- Definition of an action
- Definition of a function
- Example of a Z algorithm

4. Z language (Data structures)

- Structures
- Arrays
- Linked lists
- Bidirectional linked lists
- Queues
- Piles
- Binary search trees

M-ary search trees
Files

5. Abstract machines

Abstract machines
Arrays
Structures
Linked lists
Bidirectional linked lists
Stacks
Queues
Binary search trees
M-ary search trees
Files

6. Translation from Z to PASCAL

Declarations
Expressions
Assignment
While loop
For loop
If statement
Reading
Writing
Action
Function
Standard functions
Algorithm

7. Implementation of Z machines in PASCAL

Arrays
Structures
Linked lists
Bidirectional linked lists
Stacks
Queues
Binary search trees
M-ary search trees
Files

8. Translation from Z to C

Declarations
Expressions
Assignment
While loop
For loop
If statement
Reading
Writing
Action
Function
Standard functions

Algorithm

9. Implementation of Z machines to C

Arrays

Structures

Linked lists

Bidirectional linked lists

Stacks

Queues

Binary search trees

M-array search trees

Files

10. Keyword index

General presentation

Presentation
Menus
Traitments
Z Language
Documentation

Presentation

KHAWARIZM is an environment for learning and deepening the main data and file structures.

KHAWARIZM offers the possibility to write algorithms in an algorithmic language (Z-language), to indent them, to run or simulate them and to convert them automatically to the PASCAL and C programming languages.

KHAWARIZM aims at the assisted design of algorithms.

Menus

KHAWARIZM offers several windows showing :

- Data (Readings)
- Results of the run (Writings)
- Results of the simulation (Complete trace)

At any time in KHAWARIZM, you can invoke the help (F1) or activate operations with buttons.

Traitments

KHAWARIZM offers the following services:

- An editor to write your algorithms providing all the documentation on the Z language.
- An indenter to arrange your algorithms.

Its main features are :

- . each statement is written on a different line,
- . the keywords are rewritten in upper case (or lower case),
- . the first character of any identifier is rewritten in upper case,
- . control structures are highlighted,
- . instructions of the same level start on the same column.
- . the "ventilation step" is variable.
- An interpreter to execute your algorithms, giving as a result all the writings emitted (Results window).

- A simulator to give the complete progress (window "Simulation") of your algorithms by showing the evolution of all the manipulated objects. This helps you to correct or build your algorithms.

- A translator for automatic conversion of algorithms to Pascal or C programs.

[Z Language](#)

In KHAWARIZM, the algorithms are expressed in an algorithmic language (the Z language).

The particularity of the Z language lies in the fact that it can write algorithms on abstract machines simulating the main data structures.

The Z language is designed primarily for the following purposes:

- Experimenting on the main data structures, regardless of their implementation, by developing algorithms on
 - . arrays
 - . structures,
 - . linked lists,
 - . two-sided lists,
 - . queues,
 - . stacks,
 - . binary search trees,
 - . m-ary search trees.
- Creating and managing complex data structures including
 - . linked list of queues,
 - . linked list of stacks, ,
 - . tree of linked lists,
 - . linked list of stacks of arrays,
 - . etc...
- Writing recursive algorithms.

Thanks to the abstract machine defined on files, the Z language also allows the use of files and the construction of both simple and complex file structures.

[Documentation](#)

KHAWARIZM offers all the documentation on the Z language.

KHAWARIZM provides the Z --> PASCAL and Z --> C translations.

KHAWARIZM gives some possible implementations in PASCAL and C of the various abstract machines considered in the Z language.

All the documentation is collected in a hyper-text.

Steps to follow for the realization of a program under K H A W A R I Z

M.

Familiarization with an arbitrary algorithmic language

Editing the algorithm

Syntax check

Running

Simulation

Trace

Translation to a programming language

PASCAL or C Programmation

Example of a Z algorithm

PASCAL Equivalent

C Equivalent

[Familiarization with an arbitrary algorithmic language](#)

Learn the algorithmic language used.

Use the online help.

[Editing the algorithm](#)

Write an algorithm or correct an existing algorithm.

[Syntax check](#)

Run the arranger module.

Repeat as long as there are errors

. Correct the errors

. Relaunch the arranger module

At this point, your algorithm is well written and has been indented for you. (You can change the presentation modes of your algorithm (see "Options" in the menu)

[Running](#)

Start the execution of your algorithm

The windows then show

- the data read by your algorithm (Data button)
- the writings emitted by your algorithm (Button Results)

Your algorithm gives either the expected results or not. In this last case, launch the simulation to try to determine the logic errors.

[Simulation](#)

Run the simulation of your algorithm.

This is an execution with a trace.

The windows show

- the data read by your algorithm (Data button)
- the writings issued by your algorithm (Button Results)
- all the changes made on the objects used (Button Simulation)

You thus have the complete trace of your algorithm that you can print and analyze to detect errors.

If you want to see more closely the different steps of your algorithm, ask for a trace.

Trace

Request the simulation with trace again.

You can then follow step by step the evolution of your algorithm, exit the current loop or even the current module.

In order to avoid having a complete trace which can be long, it is possible to limit the length of the loops used in your algorithm. You can change the simulation modes (see "Options" in the menu).

Translation to a programming language

Once your algorithm is "running", it is possible to translate it automatically into PASCAL or C. Just click on the "To Pascal" or "To C" button. Two windows organized as "Tiles" are then shown. One contains your algorithm and the other the result of the translation.

You can consult the help concerning the transition to PASCAL or C.

In this help, you will find

- the Z to PASCAL and Z to C equivalents.
- All implementations of Z machines.

Khawarizm's task stops here.

PASCAL or C programming

Use the PASCAL or C compiler to finalize your program. In particular, you can add all the procedures of data entry and restitution of the results.

Example of a Z algorithm

```
{ Is a liked list included in another ? }  
LET  
L1 , L2 : LISTS ;  
Search , All : FUNCTION ( BOOLEAN ) ;  
BEGIN  
CREATE_LIST ( L1 , [ 2 , 5 , 9 , 8 , 3 , 6 ] ) ;  
CREATE_LIST ( L2 , [ 12 , 5 , 19 , 8 , 3 , 6 , 2 , 9 ] ) ;  
WRITE ( All ( L1 , L2 ) )  
END  
{ Search for a value in a linked list }  
FUNCTION Search ( L , Val ) : BOOLEAN  
LET  
L : LIST ;  
Val : INTEGER ;  
  
BEGIN  
IF L = NULL  
Search := FALSE  
ELSE  
IF CELL_VALUE ( L ) = Val  
Search := TRUE
```



```

ELSE
Search := Search ( NEXT ( L ) , Val )
ENDIF
ENDIF
END

{ Is L1 included in L2? }
FUNCTION All ( L1 , L2 ) : BOOLEAN
LET
L1 , L2 : LISTS ;

BEGIN
IF L1 = NULL
All := TRUE
ELSE
IF NOT Search ( L2 , CELL_VALUE ( L1 ) )
All := FALSE
ELSE
All := All ( NEXT ( L1 ) , L2 )
ENDIF
ENDIF
END

```

PASCAL Equivalent

```

{**-----**/
/** Translation Z to PASCAL (Standard) **/
/** By Pr. D.E ZEGOUR **/
/** E S I - Algier **/
/** Copyright 2014 **/
/**-----**}

PROGRAM My_program;

{ Implementation : LIST Of INTEGERS }

{ Linked lists }
TYPE
Typeelem_LI = INTEGER;
Pointer_LI = ^Maillon_LI; { type du champ 'Adresse' }
Maillon_LI = RECORD
Val : Typeelem_LI;
Next : Pointer_LI
END;

PROCEDURE Allocate_cell_LI ( VAR P : Pointer_LI );
BEGIN NEW(P) END;

PROCEDURE Free_LI ( P : Pointer_LI );
BEGIN DISPOSE(P) END;

PROCEDURE Ass_val_LI(P : Pointer_LI; Val : Typeelem_LI);

```

```
BEGIN P^.Val := Val END;
```

```
FUNCTION Cell_value_LI (P : Pointer_LI) : Typeelem_LI;  
BEGIN Cell_value_LI := P^.Val END;
```

```
FUNCTION Next_LI( P : Pointer_LI) : Pointer_LI;  
BEGIN Next_LI := P^.Next END;
```

```
PROCEDURE Ass_adr_LI( P, Q : Pointer_LI ) ;  
BEGIN P^.Next := Q END;
```

```
TYPE  
Typeelem_V6I = INTEGER ;  
Typetab_V6I = ARRAY[1..6] of Typeelem_V6I ;
```

```
TYPE  
Typeelem_V8I = INTEGER ;  
Typetab_V8I = ARRAY[1..8] of Typeelem_V8I ;
```

```
{ Declaration part of variables }  
VAR
```

```
L1 : Pointer_LI;  
L2 : Pointer_LI;  
T_L1 : Typetab_V6I ;  
T_L2 : Typetab_V8I ;
```

```
{ High level operations }
```

```
{ creating a linked list }  
PROCEDURE Create_list_LI ( VAR L : Pointer_LI ; Tab : Typetab_V6I ; N: INTEGER) ;  
VAR  
I : INTEGER;  
P, Q : Pointer_LI ;  
BEGIN  
L:=NIL;  
FOR I := 1 TO N DO  
BEGIN  
Allocate_cell_LI( Q ) ;  
Ass_val_LI (Q, Tab[I]);  
Ass_adr_LI (Q, NIL);  
IF L = NIL  
THEN L := Q  
ELSE Ass_adr_LI (P, Q);  
P := Q  
  
END;  
END;
```

```
{ Procedure and/or function prototypes }
```

```

FUNCTION Search (VAR L : Pointer_LI ; VAR Val : INTEGER) : BOOLEAN; FORWARD;
FUNCTION All (VAR L1 : Pointer_LI ; VAR L2 : Pointer_LI) : BOOLEAN; FORWARD;

```

```

{ Search for a value in a linked list }

```

```

FUNCTION Search (VAR L : Pointer_LI ; VAR Val : INTEGER) : BOOLEAN;

```

```

{ Declaration part of variables }

```

```

VAR

```

```

Search2 : BOOLEAN ;

```

```

_Px1 : Pointer_LI;

```

```

BEGIN

```

```

IF L = NIL THEN BEGIN

```

```

Search2 := FALSE END

```

```

ELSE

```

```

BEGIN

```

```

IF Cell_value_LI(L) = Val THEN BEGIN

```

```

Search2 := TRUE END

```

```

ELSE

```

```

BEGIN

```

```

_Px1 := Next_LI(L) ;

```

```

Search2 := Search (_Px1, Val )

```

```

END

```

```

END

```

```

;Search := Search2 ;

```

```

END;

```

```

{ Is L1 included in L2? }

```

```

FUNCTION All (VAR L1 : Pointer_LI ; VAR L2 : Pointer_LI) : BOOLEAN;

```

```

{ Declaration part of variables }

```

```

VAR

```

```

All2 : BOOLEAN ;

```

```

_Px1 : INTEGER ;

```

```

_Px2 : Pointer_LI;

```

```

BEGIN

```

```

IF L1 = NIL THEN BEGIN

```

```

All2 := TRUE END

```

```

ELSE

```

```

BEGIN

```

```

_Px1 := Cell_value_LI(L1) ;

```

```

IF NOT Search ( L2 ,_Px1) THEN BEGIN

```

```

All2 := FALSE END

```

```

ELSE

```

```

BEGIN

```

```

_Px2 := Next_LI(L1) ;

```

```

All2 := All (_Px2, L2 )

```

```

END

```

```

END

```

```

;All := All2 ;

```

```

END;

```

```

{ Body of main program }
BEGIN
T_L1 [ 1 ] := 2 ;
T_L1 [ 2 ] := 5 ;
T_L1 [ 3 ] := 9 ;
T_L1 [ 4 ] := 8 ;
T_L1 [ 5 ] := 3 ;
T_L1 [ 6 ] := 6 ;
Create_list_LI ( L1 , T_L1 , 6 ) ;
T_L2 [ 1 ] := 12 ;
T_L2 [ 2 ] := 5 ;
T_L2 [ 3 ] := 19 ;
T_L2 [ 4 ] := 8 ;
T_L2 [ 5 ] := 3 ;
T_L2 [ 6 ] := 6 ;
T_L2 [ 7 ] := 2 ;
T_L2 [ 8 ] := 9 ;
Create_list_LI ( L2 , T_L2 , 8 ) ;
WRITELN ( All(L1,L2) )
;READLN;
END.

```

C Equivalent

```

/**-----**/
/** T r a n s l a t i o n Z to C (Standard) **/
/** By Pr. D.E ZEGOUR **/
/** E S I - Algier **/
/** Copyright 2014 **/
/**-----**/

```

```

/* Is a liked list included in another ? */

```

```

#include <stdio.h>
#include <stdlib.h>

```

```

typedef int bool ;

```

```

#define True 1
#define False 0

```

```

/** -Implementation- **\: LIST Of INTEGERS**/

```

```

/** Linked lists **/

```

```

typedef int Typeelem_Li ;
typedef struct Maillon_Li * Pointer_Li ;

```

```

struct Maillon_Li
{
Typeelem_Li Val ;

```

```

Pointer_Li Next ;
} ;

Pointer_Li Allocate_cell_Li (Pointer_Li *P)
{
    *P = (struct Maillon_Li *) malloc( sizeof( struct Maillon_Li)) ;
    (*P)->Next = NULL;
}

void Ass_val_Li(Pointer_Li P, Typeelem_Li Val)
{
    P->Val = Val ;
}

void Ass_adr_Li( Pointer_Li P, Pointer_Li Q)
{
    P->Next = Q ;
}

Pointer_Li Next_Li( Pointer_Li P)
{ return( P->Next ) ; }

Typeelem_Li Cell_value_Li( Pointer_Li P)
{ return( P->Val) ; }

void Free_Li ( Pointer_Li P)
{ free (P);}

/** For temporary variables **/
typedef int Typeelem_V6i;
typedef Typeelem_V6i Typetab_V6i[6];

/** For temporary variables **/
typedef int Typeelem_V8i;
typedef Typeelem_V8i Typetab_V8i[8];

/** Variables of main program **/
Pointer_Li L1=NULL;
Pointer_Li L2=NULL;
Typetab_V6i T_L1;
Typetab_V8i T_L2;

/** High level operations **/

/** creating a linked list **/
void Create_list_Li ( Pointer_Li *L, Typetab_V6i Tab, int N)
{
    int I ;
    Pointer_Li P, Q ;

    *L =NULL;

```

```

for( I=1;I<=N;++I)
{
Allocate_cell_Li( &Q ) ;
Ass_val_Li (Q, Tab[I-1]);
Ass_adr_Li (Q, NULL);
if (*L == NULL)
*L = Q ;
else Ass_adr_Li (P, Q);
P = Q ;
}
}

/** Function prototypes */

bool Search (Pointer_Li *L , int *Val) ;
bool All (Pointer_Li *L1 , Pointer_Li *L2) ;

/* Search for a value in a linked list */
bool Search (Pointer_Li *L , int *Val)
{
/** Local variables */
bool Search2 ;
Pointer_Li _Px1=NULL;

/** Body of function */
if( *L == NULL) {
Search2 = False; }
else
{
if( Cell_value_Li ( *L ) == *Val) {
Search2 = True; }
else
{
_Px1 = Next_Li ( *L ) ;
Search2 = Search ( &_Px1, & *Val );
}
}
return Search2 ;
}

/* Is L1 included in L2? */
bool All (Pointer_Li *L1 , Pointer_Li *L2)
{
/** Local variables */
bool All2 ;
int _Px1;
Pointer_Li _Px2=NULL;

/** Body of function */
if( *L1 == NULL) {
All2 = True; }
else

```

```

{
_Px1 = Cell_value_Li ( *L1 );
if( ! Search ( & *L2 , & _Px1)) {
All2 = False; }
else
{
_Px2 = Next_Li ( *L1 );
All2 = All ( & _Px2, & *L2 );
}
}
return All2 ;
}

int main(int argc, char *argv[])
{
T_L1 [ 0 ] = 2 ;
T_L1 [ 1 ] = 5 ;
T_L1 [ 2 ] = 9 ;
T_L1 [ 3 ] = 8 ;
T_L1 [ 4 ] = 3 ;
T_L1 [ 5 ] = 6 ;
Create_list_Li (&L1 , T_L1 , 6 );
T_L2 [ 0 ] = 12 ;
T_L2 [ 1 ] = 5 ;
T_L2 [ 2 ] = 19 ;
T_L2 [ 3 ] = 8 ;
T_L2 [ 4 ] = 3 ;
T_L2 [ 5 ] = 6 ;
T_L2 [ 6 ] = 2 ;
T_L2 [ 7 ] = 9 ;
Create_list_Li (&L2 , T_L2 , 8 );
printf ( " %d", All(&L1,&L2) );

system("PAUSE");
return 0;
}

```

Z Language (Basic)

-

Overview
Structure of a Z algorithm
Control structures
Simple actions
Scalars
Pointers
Expressions
Comments
High level operations
Standard functions
Fonctions de génération aléatoire
Functions on strings
Definition of an action
Definition of a function
Example of a Z algorithm

Overview

- > A Z algorithm is a set of modules. The first one is the main module and the others are either actions (ACTION) or functions (FUNCTION).
- > The Z language accepts recursion.
- > Static objects are declared in the main module.
- > The communication between modules is made via parameters and static variables.
- > The language allows:
 - Any type of parameters : scalars, structures, linked lists , queues, stacks, arrays, trees and also complex types.
 - The dynamic allocation of arrays and structures
 - The global assignment of any type
- > Four standard types (scalars) are allowed : **INTEGER, BOOLEAN, CHAR, STRING** .
- > Some usual functions exist : **MOD, MAX, MIN, RANDNUMBER, RANDSTRING**,...
- > The language is the set of abstract algorithms written by using abstract machines.
- > So, we consider abstract machines on structures, arrays of any dimension, queues, stacks, binary and m-ary search trees, linked lists and bidirectional linked lists.
- > We also consider an abstract machine on the files allowing their use and the construction of simple structures of files as well as the most complex structures.
- > The language allows compound types such as **STACK OF QUEUE OF LISTS OFOF** which the last one quoted is of scalar type or simple structure.
- > The language has high-level operations to build lists, trees, queues, etc. from a set of values (expressions) or structures.
- > the language offers two very useful functions to randomly generate strings (**ALEACHAINE**) and integers (**ALEANUMBER**).
- > The language allows reading and writing scalars, n-dimensional arrays of scalars and simple or complex structures.

Structure of a Z algorithm

```
LET
  { Local and static objects }
  { announcement of the modules }
```

```
BEGIN
  { Statements }
```

```
END
```

```
Module 1
```

```
....
```

```
Module n
```

Each module can be either a function or an action.

Control structures

While loop

```
WHILE E [ : ]
  { Statements }
ENDWHILE
```

For loop

```
FOR V := E1, E2 [,E3]
  { Statements }
ENDFOR
```

E3, if present, designates the step.

Conditional

```
IF E [ : ]
  { Statements }
[ ELSE
  { Statements } ]
ENDIF
```

Simple actions

V denotes a variable, E an expression and Idf a module identifier.

Assignment : V := E

Reading : READ(V1, V2,)

Variables can be scalars, structures or arrays of any dimension.

Writing : WRITE(E1, E2,)

Expressions can be scalars, structures or n-dimensional arrays.

Scalars

Quatre types standards sont autorisés : INTEGER, CHAR, STRING, BOOLEAN.

Definition of scalars

```
LET <li> Sep Type
```

 : identifier list separated by coma.

Type is a standard type.

Examples

```
A, B, Found : BOOLEANS ;  
A : INTEGER ;  
X, Y, Z : STRINGS ;
```

Pointers

We define pointer variables to use data structures.

```
LET <li> : POINTER TO Typec [ OF Typec OF Typec OF .... ] [ OF Types]
```

 : Identifier list separated by coma.

Typec in { **ARRAY**, **STACK**, **LIST**, **QUEUE**, **BST**, **BILIST**, **MST** }

Types is a scalar or simple structure.

Note

We can do not use the pointer type. Indeed, declarations

"**LET L : LIST**" and

"**LET L : POINTER TO LIST**"

are equivalent.

Examples

```
P1 : POINTER TO LIST OF STACKS;  
P2, P2 : POINTERS TO BST;
```

Z Expressions

As in the programming languages.

Arithmetical expressions : + , - , / , *

Logical expressions : **AND**, **OR**, **NOT**

Expressions on strings : +

Relational expressions : < , <= , > , >= , = , <> (ou #)

Logical constants : **TRUE**, **FALSE**

Pointer constant : **NULL**

Examples

```
B+C / F  
NOT Found  
(X # 5) AND NOT Found  
F(X) <> 5  
P = Null
```

Comments

Comments are inserted into any place where we can insert a space.

Comments are between { and } or between /* and */.

High level operations

The Z language has high-level operations for filling or initializing a data structure from a set of values.

CREATE_ARB, CREATE_LISTE, CREATE_LISTEBI, CREATE_ARM, CREATE_PILE, CREATE_FILE, INIT_VECT (or INIT_ARRAY), INIT_STRUCT.

Syntax :

```
CREATE_LIST ( L, [Exp1, Exp2, ....] )
CREATE_BILIST ( LB, [Exp1, Exp2, ....] )
CREATE_BST ( A, [Exp1, Exp2, ....] )
CREATE_MST ( M, [Exp1, Exp2, ....] )
CREATE_QUEUE ( F, [Exp1, Exp2, ....] )
CREATE_STACK ( P, [Exp1, Exp2, ....] )
INIT_STRUCT(S, [Exp1, Exp2, ....])
INIT_ARRAY( T, [Exp1, Exp2, ....] )
```

Exp1, Exp2, are scalar expressions or structures.

Examples

```
CREATE-LIST (L, [12, 23, 67, I, I+J] )
```

creates the linked list L with values with the values in square brackets in the order shown.

If L1 and L2 are 2 integer linked list and L1 a linked list of a linked list of integers, we can then write :

```
CREATE_LIST ( L1 , [ 2 , 4 , 67 , 778 ] ) ;
CREATE_LIST ( L2 , [ 12 , 14 , 167 , 1778 ] ) ;
CREATE_LIST ( L1 , [ L1 , L2 ] ) ;
```

Standard functions

MOD (A, B) : Give the remainder of the integer division of A by B.

MAX(A, B) : Give the maximum between A and B.

MIN(A, B) : Give the minimum between A and B.

EXP(A, B) : A exposant B

Fonctions de génération aléatoire

RANDSTRING(N) : Generate an alphanumeric string of N characters.

RANDNUMBER (N) : Generate an integer between 0 and N.

Functions on strings

CHARACT(S, I) : give the i-th character of the string S.

LENTHSTRING(S) : give the length of string S

Definition of an action

```
ACTION Name (P1, P2, ..., Pn)
    { Local objects and parameters }
BEGIN
    { Statements }
END
```

Calling an action is made by the operation CALL followed by the name of the action and its parameters.

Parameters are not protected by the action.

Definition of a function

```
FUNCTION Name (P1, P2, ...,Pn) : Type
    { Local objects and parameters }
BEGIN
    { Statements }
END
```

Type can be any.

Functions are used in expressions.

Parameters are not protected by the function.

Example of a Z algorithm

```
LET
L1, L2 : LISTS;
Rech, Tous : FUNCTION(BOOLEAN);
BEGIN
CREATE_LIST(L1, [2, 5, 9, 8, 3, 6 ]);
CREATE_LIST(L2, [12, 5, 19, 8, 3, 6, 2,9]);
WRITE( Tous(L1, L2) )
END
```

```
FUNCTION Rech ( L, Val ) : BOOLEAN
LET
L : LIST; Val : INTEGER;
BEGIN
IF L = NULL : Rech := FALSE
ELSE
IF CELL_VALUE(L) = Val :
Rech := TRUE
ELSE
Rech := Rech(NEXT(L), Val )
ENDIF
ENDIF
END
```

```
FUNCTION Tous ( L1, L2 ) : BOOLEAN
LET
L1, L2 : LISTS;
BEGIN
IF L1 = NULL : Tous := TRUE
ELSE
IF NOT Rech(L2, CELL_VALUE(L1) )
Tous := FALSE
ELSE
Tous := Tous(NEXT(L1), L2)
ENDIF
ENDIF
END
```

Z Language (Data structures)

Structures
Arrays
Linked lists
Bidirectional linked lists
Queues
Piles
Binary search trees
M-ary search trees
Files

Structures

A structure is a set of heterogeneous elements. An element of a structure can be a scalar or a one-dimensional array of scalars.

a structure can be complex, i.e., consisting of scalars and/or one dimensional array of scalars.

Structures can be static or dynamic.

Structure declaration:

LET : [**STRUCTURE**] (Type1, type2, ...) [**DYNAMIC**]

 : identifier list separated by coma.

Typei is either a scalar type or a one-dimensional array of scalars.

Examples

S1 : (INTEGER, STRING) ;

S2 : STRUCTURE (CHAR, INTEGER, BOOLEAN) ;

S3 : (INTEGER, ARRAY(5) OF STRING) DYNAMIC;

Arrays

An array is a set of heterogeneous elements.

An array can be simple, i.e., consisting only of scalars.

An array can be complex, i.e., consisting of simple structures.

An array can be static or dynamic.

Array declaration:

LET : **ARRAY** (Dim1, Dim2, ...) **OF** Typec **OF** Typec **OF** **OF** Types [**DYNAMIC**]

 : identifier list separated by coma.

Typec dans { **ARRAY**, **STACK**, **LIST**, **QUEUE**, **BST**, **BILIST**, **MST** }

Types is a scalar or a simple type.

Examples

V1 : ARRAY (5) DYNAMIC;

V2, V3 : **ARRAYS** (3, 8) **OF STRINGS** ;

Linked lists

A linked linear list is a set of chains that are dynamically allocated (i.e. during the execution of a program) and linked together. A chain generally contains two fields: information and an address. It is the user who creates the list, who modifies it by adding or deleting chains, who browses it in order to perform a given treatment.

Champ 'Value' can be of any type.

Linked list declaration:

LET : **LISTE** [**OF** Typec **OF** Typec **OF**] [**OF** Types]

 : identifier list separated by coma.

Typec dans { **ARRAY**, **STACK**, **LIST**, **QUEUE**, **BST**, **BILIST**, **MST** }

Types is a scalar or a simple type.

Examples

L : **LIST OF** (**STRING**, **INTEGER**);

L : **LIST OF STACKS OF STRINGS** ;

L : **LIST OF STRINGS**;

Bidirectional linked lists

A bilateral list is a linked linear list that can be traversed in both directions.

An element has three fields: value, left address and right address.

Champ 'Value' can be of any type.

Bidirectional linked list declaration:

LET : **LISTEBI** [**OF** Typec **OF** Typec **OF**] [**OF** Types]

 : identifier list separated by coma.

Typec dans { **ARRAY**, **STACK**, **LIST**, **QUEUE**, **BST**, **BILIST**, **MST** }

Types is a scalar or a simple type.

Examples

Lb1 : **BILIST OF** (**STRING**, **INTEGER**);

Lb2 : **BILIST OF STACKS OF STRINGS** ;

Lb3 : **BILIST OF STRINGS**;

Queues

A queue obeys the 'FIFO' principle: First In First Out. It is a collection of elements such that any new element is inserted at the end and any removal of element is done at the beginning.

An element can be of any type.

Queue declaration:

LET : **QUEUE** [**OF** Typec **OF** Typec **OF**] [**OF** Types]

 : identifier list separated by coma.

Typec dans { **ARRAY**, **STACK**, **LIST**, **QUEUE**, **BST**, **BILIST**, **MST** }

Types is a scalar or a simple type.

Examples

```
F1 : QUEUE OF (STRING, INTEGER);  
F2 : QUEUE OF STACKS OF STRINGS ;  
F3 : QUEUE OF STRINGS;
```

Stacks

A stack obeys the 'LIFO' principle: Last In First Out. It is a collection of elements such that any new element is inserted at the beginning and any removal of element is done at the beginning.

An element can be of any type.

Stack declaration

```
LET <Li> : STACK [OF Typec OF Typec OF .... ][ OF Types]
```

 : identifier list separated by coma.

Typec dans { ARRAY, STACK, LIST, QUEUE, BST, BILIST, MST }

Types is a scalar or a simple type.

Examples

```
P1 ; STACK OF (STRINGS, INTEGER);  
P2 ; STACK OF STACKS OF STRINGS ;  
P3 ; STACK OF STRINGS;
```

Binary search trees

A binary search tree represents a set of data with an order relation.

All data in the left sub-tree of any node with the information x are less than x, all data in the right sub-tree of any node with the information x are greater than x.

A node can be of any type.

Binary search tree declaration:

```
LET <Li> : BST [OF Typec OF Typec OF .... ][ OF Types]
```

 : identifier list separated by coma.

Typec dans { ARRAY, STACK, LIST, QUEUE, BST, BILIST, MST }

Types is a scalar or a simple type.

Examples

```
A1 : BST OF (STRING, INTEGER);  
A2 : BST OF STACKS OF STRINGS ;  
A3 : BST OF STRINGS;
```

M-ary search trees

An m-ary search tree allows to represent a data set with an order relation. It is a generalization of the binary search tree. Instead of having one information and two address fields, we have p addresses and (p-1) information in which case the m-ary search tree is said to have order p.

Intuitively, a node has the following form: (a1, v1, a2, v2,, vp-1, ap). All the data in the sub-

tree of root a1 are less than v1, all the data in the sub-tree of root a2 are greater than v1 and less than v2, etc.

A node can be of any type.

M-ary search tree declaration:

LET : **MST** (degree) [**OF** Typec **OF** Typec **OF**] [**OF** Types]

 : identifier list separated by coma.

Typec dans { **ARRAY**, **STACK**, **LIST**, **QUEUE**, **BST**, **BILIST**, **MST** }

Types is a scalar or a simple type.

Examples

M1 : **MST**(4) **OF** (**STRING**, **INTEGER**);
M2 : **MST**(2) **OF** **STACKS OF STRINGS** ;
M3 : **MST**(3) **OF** **STRINGS**;

Files

A file is a set of records (structures) generally stocked in disks.

For a designer, a file is a set of blocks.

The file holds a special block (Header block) for the design of file structures.

.

File declaration

LET File_name : **FILE OF** Type [**HEADER** (Type1, Type2,)] **BUFFER**

 : identifier list separated by coma.

A file definition consists of 3 parts:

** The first part (**FILE**) specifies the nature of the elements in the file.

a file record can be :

- a scalar, i.e of a simple type (**INTEGER**, **BOOLEAN**, **CHAR**, **STRING**),
- a one-dimensional array of scalars
- a structure that can contain scalars or one-dimensional arrays of scalars.

** The second part (**HEADER**) describes the file features by specifying the type of each feature.

This part is mainly used for the creation of user file structures and is used to store all the information useful for the exploitation of the file.

** The third part (**BUFFER**) defines the variables (record or block) used in reading and writing operations.

Examples

F1 : **FILE OF CHAINES BUFFER** V1, V2;
F2 : **FILE OF VECTEUR(5) OF ENTIER BUFFER** V ;
F3 : **FILE OF (INTEGER, ARRAY(3) OF CHAR) HEADER(INTEGER, INTEGER) BUFFER**
V ;
F4 : **FILE OF CHAR HEADER (INTEGER, STRING, BOOLEAN) BUFFER** V ;

Machines abstraites

Abstract machines
Arrays
Structures
Linked listes
Bidirectional linked lists
Stacks
Queues
Binary search trees
M-ary search trees
Files

Abstract machines

To each data structure an abstract machine is defined with its set of operation.

LET : <Abstract machine>

 : Identifier list separated by coma.

Examples

L1, L2 : **LISTS**;
F : **QUEUE**;
V1 : **ARRAY**(10, 60);
Y : **LIST OF STACKS OF ARRAY** (5);

Abstract machine on arrays

ELEMENT (T [i, j, ...])
Access to element T[i, j, ...] of array T.
ASS_ELEMENT (T [I, J, ...], Val)
Assign value Val to element T[i, j, ...].
ALLOC_ARRAY (T)
Array allocation. The array address is in T.
LIBER_ARRAY (T)
Free the array of address T.

Abstract machine on structures

STRUCT(S, i) :
Access to the i-th field of structure S.
ASS_STRUCT(S, i, Val) :
Assign value Val to the i-th field of structure S.
ALLOC_STRUCT(S) :
Allocation of a structure. The structure address is in S.
FREE_STRUCT(S) :

Free the structure of address S.

Abstract machine on linked lists

ALLOCATE (P) :

Create a chain and return its address in P.

FREE (P) :

Free the node of address P.

NEXT (P) :

Access to the 'Address' field of the node referenced by P.

VALUE (P) :

Access to the 'Value' field of the node referenced by P.

ASS_ADR (P, Q) :

Assign to the 'Address' field of the node referenced by P, the address Q.

ASS_VAL(P, Val) :

Assign to the 'Value' field of the node referenced by P, the value Val.

Abstract machine on bidirectional linked lists

ALLOCATE (P) :

Create a chain and return its address in P.

FREE (P) :

Free the node of address P.

NEXT (P) :

Access to the 'Right address' field of the node referenced by P.

PREVIOUS (P) :

Access to the 'Left address' field of the node referenced by P.

VALUE(P) :

Access to the 'Value' field of the node referenced by P.

ASS_VAL(P, Val) :

Assign to the 'Value' field of the node referenced by P, the value Val.

ASS_R_ADR (P, Q) :

Assign to the 'Right address' field of the node referenced by P, the address Q.

AFF_L_ADR (P, Q) :

Assign to the 'Left address' field of the node referenced by P, the address Q.

Abstract machine on queues

CREATEQUEUE (F) :

Create an empty queue.

EMPTY_QUEUE (F) :

Test if queue F is empty.

ENQUEUE (F, Val) :

Enqueue (add to tail) value Val to queue F.

DEQUEUE (F, Val) :

Dequeue (Remove from the file head) a value from queue F et put it in Val.

Abstract machine on stacks

CREATESTACK (P) :

Create an empty stack.

EMPTY_STACK (P) :

Test if stack P is empty.

PUSH (P, Val) :

Push (add to the top) value Val to stack P.

POP (P, Val) :

Pop(Remove from the top) a value from stack P and put it in Val.

Abstract machine on binary search trees

ALLOCATE_NODE (Val) :

Create a node value Val and return the address of the node. The others fields are to NIL.

FREE_NODE (P) :

Free the node of address P.

LC (P) :

Access to field left child of node referenced by P.

RC (P) :

Access to field right child of node referenced by P.

PARENT (P) :

Access to field parent of node referenced by P.

NODE_VALUE (P) :

Access to field value of node referenced by P.

ASS_LC (P, Q) :

Assign the address Q to the field left child of node referenced by P.

ASS_RC (P, Q) :

Assign the address Q to the field rightchild of node referenced by P.

ASS_PARENT (P, Q) :

Assign the address Q to the field parent of node referenced by P.

ASS_NODE_VAL(P, Val) :

Assign the value Val to the field value of node referenced by P.

Abstract machine on M-ary search trees

ALLOCATE_NODE (Val) :

Create a node value Val and return the address of the node. The others fields are to NIL.

FREE_NODE (P) :

Free the node of address P.

CHILD (P, I) :

Access to the i-th child of node referenced by P.

PARENT (P) :

Access to field parent of node referenced by P.

NODE_VALUE_MST (P, I) :

Access to the i-th value of node referenced by P.

ASS_CHILD (P, I, Q) :

Assign the address Q to the i-th child of node referenced by P.

ASS_PARENT (P, Q) :

Assign the address Q to the field parent of node referenced by P.

ASS_NODE_VAL_MST(P, I, Val) :

Assign the value Val to i-th value of node referenced by P, la valeur Val.

DEGREE(P) :

Number of values stocked inside the node referenced by P.

ASS_DEGREE(P, n) :

Assign the value n to the field degree of node referenced by P.

Abstract machine on files

OPEN (F, Fp, Mode) :

Open the logical file F and associate it to the physical file specifying whether the file is new ('N') or old ('A').

CLOSE(F) :

Close file F.

READSEQ (F, V) :

Read in buffer V the record (or block) that is in current position.

WRITESEQ (F, V) :

Write the record (or block) V to the current position.

READDIR (F, V, n) :

Read into V the n-th record (or block) of file F.

WRITEDIR (F, V, n) :

Write record (or block) V to n-th position.

ADD (F, V) :

Add record (or block) V at the end of file F.

ENDFILE (F) :

Predicate equal to true if the end of file F is reached, false otherwise.

ALLOC_BLOCK(F) :

Position of a block in which we can write.

HEADER (F, i) :

Get the i-th feature of file F.

ASS_HEADER(F, i, v) :

Assign V as the i-th feature of file F.

Translation from Z to PASCAL

Declarations
Expressions
Assignment
While loop
For loop
If statement
Reading
Writing
Action
Function
Standard functions
Algorithm

Variable declaration

A variable declaration in C is made by

 Type ;

where denotes an identifier list.

LET is translated to VAR.

Simple objects

Equivalents Z --> PASCAL
 Z PASCAL

INTEGER INTEGER
BOOLEAN BOOLEAN

Expressions

Z-expression grammar is included in PASCAL grammar.

Assignment

Same syntax

WHILE loop

-----Z-----
WHILE Cond :
 Statements
ENDWHILE

is translated to:

```
-----PASCAL-----
WHILE ( Cond ) DO
  BEGIN
    Statements
  END
```

FOR loop

```
-----Z-----
FOR V:= Exp1, Exp2 [, Exp3] :
  Statements
ENDFOR
```

if Exp3 is absent or equal to 1

is translated to:

```
-----PASCAL-----
FOR V:= Exp1 TO Exp2 DO
  BEGIN
    Instructions
  END
```

If Exp3 # 1 :

```
-----PASCAL-----
V := Exp1;
WHILE ( V <= Exp2 ) DO
  BEGIN
    Statements ;
    V := V + Exp3
  END;
```

If statement

```
-----Z-----
IF Cond :
  Statements
[ELSE
  Statements]
ENDIF
```

is translated to:

```
-----PASCAL-----
IF Cond
THEN
  BEGIN
    Statements
  END
[ELSE
  BEGIN
```

Statements
END]

Reading

-----Z-----
READ(V1, V2, ...)

is translated to:

-----PASCAL-----
READLN(V1, V2, ...)

Writing

-----Z-----
WRITE(Exp1, Exp2, ...)

is translated to:

-----PASCAL-----
WRITELN(Exp1, Exp2, ...)

Actions

-----Z-----
ACTION Name (P1, P2, ...)
LET
 Declaration of local objects and parameters
BEGIN
 Statements
END

is translated to:

-----PASCAL-----
PROCEDURE Name (VAR P1: typ; VAR P2:typ, ...);
VAR
 Declaration of local objects and parameters
BEGIN
 Statements
END

Functions

-----Z-----
FUNCTION Name (P1, P2, ...) : Type
LET
 Declaration of local objects and parameters
BEGIN
 Statements
END

is translated to:

```
-----PASCAL-----  
FUNCTION Name ( VAR P1: typ; VAR P2:typ, ...) : Type;  
VAR  
    Declaration of local objects and parameters  
BEGIN  
    Statements  
END
```

Standard functions

MOD (a, b)

```
FUNCTION Mod (a, b : INTEGER) : INTEGER;  
BEGIN  
    Mod := a Mod b  
END;
```

MIN (a, b)

```
FUNCTION Min (a, b: INTEGER) : INTEGER;  
BEGIN  
    Min := a; IF b < a THEN Min := b;  
END;
```

MAX (a, b)

```
FUNCTION Max (a, b: INTEGER) : INTEGER;  
BEGIN  
    Max := a; IF b > a THEN Max := b;  
END;
```

EXP (a, b)

```
FUNCTION Exp (a, b: INTEGER) : INTEGER;  
VAR I : INTEGER;  
BEGIN  
    Exp := 1;  
    FOR I:= 1 TO b DO Exp := Exp * a  
END;
```

RANDNUMBER (N)

```
FUNCTION Randnumber (N: INTEGER) : INTEGER;  
BEGIN  
    Randnumber := Random( N );  
END;
```

RANDSTRING (N)

```
FUNCTION Randstring(N: INTEGER) : STRING;
```



```

VAR
  K : BYTE;
  Chaine : STRING;
BEGIN
  Chaine := "";
  FOR K:=1 TO N DO
    CASE Random(2) OF
      0 : Chaine := Chaine + CHR(97+Random(26) ) ;
      1 : Chaine := Chaine + CHR(65+Random(26) )
    END;
  Randstring := Chaine;
END;

```

STRINGLENGTH (C)

```

FUNCTION Stringlength(C : STRING): INTEGER;
BEGIN
  Stringlength := length (C)
END;

```

Algorithm

```

-----Z-----
LET
  Local and static objects
  Announcement of the modules
BEGIN
  Statements
END
Module 1
Module 2
...
Modules n

```

is translated to:

```

-----PASCAL-----
PROGRAM Pascal;
VAR
  Local and static objects
  { Definition of modules }
Module 1
Module 2
...
Module n
BEGIN
  Statements
END.

```

Implementation of Z machines in PASCAL

Arrays
Structures
Linked lists
Bidirectional linked lists
Stacks
Queues
Binary search trees
M-ary search trees
Files

Array implementation in PASCAL

LET Tab : ARRAY (5 , 10) ;

{ -Implementation- : ARRAY OF INTEGERS }

{ Arrays }

TYPE

Typeelem_V5_10I = INTEGER;

Typetab_V5_10I = ARRAY[1..5,1..10] OF Typeelem_V5_10I;

Typevect_V5_10I = ^ Typetab_V5_10I;

FUNCTION Element_V5_10I (V:Typevect_V5_10I; I1 , I2 : INTEGER) : Typeelem_V5_10I ;

BEGIN

Element_V5_10I := V^[I1 ,I2];

END;

PROCEDURE Ass_element_V5_10I (V :Typevect_V5_10I; I1 , I2 :INTEGER; Val :

Typeelem_V5_10I);

BEGIN

V^[I1 ,I2] := Val;

END;

{ Declaration part of variables }

VAR

Tab : Typevect_V5_10I;

{ Body of main program }

BEGIN

NEW(Tab);

END.

Structure implementation in PASCAL

LET S : STRUCTURE (STRING , INTEGER) ;

```
TYPE Typestring = STRING[255];
```

```
{ Structures }
```

```
TYPE
```

```
Type1_TSI = Typestring;
```

```
Type2_TSI = INTEGER;
```

```
Typestr_TSI = ^ Type_TSI ;
```

```
Type_TSI = record
```

```
Field1 : Type1_TSI ;
```

```
Field2 : Type2_TSI ;
```

```
END;
```

```
FUNCTION Struct1_TSI ( S: Typestr_TSI) : Type1_TSI;
```

```
BEGIN
```

```
STRUCT1_TSI := S^.Field1;
```

```
END;
```

```
FUNCTION Struct2_TSI ( S: Typestr_TSI) : Type2_TSI;
```

```
BEGIN
```

```
STRUCT2_TSI := S^.Field2;
```

```
END;
```

```
PROCEDURE Ass_struct1_TSI ( S: Typestr_TSI; Val :Type1_TSI );
```

```
BEGIN
```

```
S^.Field1 := Val;
```

```
END;
```

```
PROCEDURE Ass_struct2_TSI ( S: Typestr_TSI; Val :Type2_TSI );
```

```
BEGIN
```

```
S^.Field2 := Val;
```

```
END;
```

```
{ Declaration part of variables }
```

```
VAR
```

```
S : Typestr_TSI;
```

```
{ Body of main program }
```

```
BEGIN
```

```
NEW(S);
```

```
END.
```

[Linked list implementation in PASCAL](#)

LET L : LIST ;

```
{ -Implementation- : LIST Of INTEGERS }
```

```
{ Linked lists }
```

```
TYPE
```

```
Typeelem_LI = INTEGER;
```

```
Pointer_LI = ^Maillon_LI; { type du champ 'Adresse' }
```

```
Maillon_LI = RECORD
```

```
Val : Typeelem_LI;
```

```

Next : Pointer_LI
END;

PROCEDURE Allocate_cell_LI ( VAR P : Pointer_LI );
BEGIN NEW(P) END;

PROCEDURE Free_LI ( P : Pointer_LI );
BEGIN DISPOSE(P) END;

PROCEDURE Ass_val_LI(P : Pointer_LI; Val : Typeelem_LI );
BEGIN P^.Val := Val END;

FUNCTION Cell_value_LI (P : Pointer_LI) : Typeelem_LI;
BEGIN Cell_value_LI := P^.Val END;

FUNCTION Next_LI( P : Pointer_LI) : Pointer_LI;
BEGIN Next_LI := P^.Next END;

PROCEDURE Ass_adr_LI( P, Q : Pointer_LI );
BEGIN P^.Next := Q END;

{ Declaration part of variables }
VAR
L : Pointeur_LI;
{ Body of main program }
BEGIN
END.

```

Bidirectional linked list implementation in PASCAL

LET L : BILIST ;

```

{ -Implementation- : BIDIRECTIONAL LIST OF INTEGERS}

{ Bidirectional linked lists }
TYPE
Typeelem_RI = INTEGER;
Pointer_RI = ^Maillon_RI; { type du champ 'Adresse' }
Maillon_RI = RECORD
Val : Typeelem_RI;
Next : Pointer_RI;
Prev : Pointer_RI
END;

PROCEDURE Allocate_cell_RI ( VAR P : Pointer_RI );
BEGIN NEW(P) END;

PROCEDURE Free_RI ( P : Pointer_RI );
BEGIN DISPOSE(P) END;

PROCEDURE Ass_val_RI (P : Pointer_RI; Val : Typeelem_RI );
BEGIN P^.Val := Val END;

```

```

FUNCTION Cell_value_RI ( P : Pointer_RI ) : Typeelem_RI;
BEGIN Cell_value_RI := P^.Val END;

```

```

FUNCTION Next_RI( P : Pointer_RI ) : Pointer_RI;
BEGIN Next_RI := P^.Next END;

```

```

FUNCTION Previous_RI( P : Pointer_RI ) : Pointer_RI;
BEGIN Previous_RI := P^.Prev END;

```

```

PROCEDURE Ass_r_adr_RI( P, Q : Pointer_RI ) ;
BEGIN P^.Next := Q END;

```

```

PROCEDURE Ass_l_adr_RI( P, Q : Pointer_RI ) ;
BEGIN P^.Prev := Q END;

```

```

{ Declaration part of variables }
VAR
Lb : Pointeur_RI;
{ Body of main program }
BEGIN
END.

```

Binary search tree implementation in PASCAL

LET A : BST ;

{ -Implementation- : BINARY SEARCH TREE OF INTEGERS }

```

{ Binary search trees }
TYPE
Typeelem_AI = INTEGER;
Pointer_AI = ^Noeud_AI;
Noeud_AI = RECORD
Element : Typeelem_AI;
Lc, Rc, Parent : Pointer_AI ;
END;

```

```

FUNCTION Node_value_AI(P : Pointer_AI) : Typeelem_AI;
BEGIN Node_value_AI := P^.Element END;

```

```

FUNCTION Lc_AI( P : Pointer_AI) : Pointer_AI;
BEGIN Lc_AI := P^.Lc END;

```

```

FUNCTION Rc_AI( P : Pointer_AI) : Pointer_AI;
BEGIN Rc_AI := P^.Rc END;

```

```

FUNCTION Parent_AI( P : Pointer_AI) : Pointer_AI;
BEGIN Parent_AI := P^.Parent END;

```

```

PROCEDURE Ass_node_val_AI ( P : Pointer_AI; Val : Typeelem_AI);
BEGIN P^.Element := Val END;

```

```

PROCEDURE Ass_lc_AI( P : Pointer_AI; Q : Pointer_AI);
BEGIN P^.Lc := Q END;

PROCEDURE Ass_rc_AI( P : Pointer_AI; Q : Pointer_AI);
BEGIN P^.Rc := Q END;

PROCEDURE Ass_parent_AI( P : Pointer_AI; Q : Pointer_AI);
BEGIN P^.Parent := Q END;

PROCEDURE Allocate_node_AI( VAR P : Pointer_AI ) ;
BEGIN
NEW ( P ) ;
P^.Lc := Nil;
P^.Rc := Nil;
P^.Parent := Nil;
END;

PROCEDURE Free_node_AI( P : Pointer_AI);
BEGIN
DISPOSE ( P )
END;

{ Declaration part of variables }
VAR
A : Pointeur_AI;
{ Body of main program }
BEGIN
END.

```

M-ary search tree implémentation in PASCAL

LET M : MST (4) ;

{ -Implementation- : M-ARY SEARCH TREE OF INTEGERS }

```

{ M-ary search trees }
TYPE
Typeelem_M4I = INTEGER;
Pointer_M4I = ^Noeud_M4I;
Noeud_M4I = RECORD
Infor : ARRAY[1..4] of Typeelem_M4I;
Child : ARRAY[1..4] of Pointer_M4I;
Degree : Byte ;
Parent : Pointer_M4I
END;

FUNCTION Node_value_mst_M4I(P : Pointer_M4I; I : INTEGER) : Typeelem_M4I;
BEGIN Node_value_mst_M4I := P^.Infor[I] END;

FUNCTION Child_M4I( P : Pointer_M4I; I : INTEGER) : Pointer_M4I;
BEGIN Child_M4I := P^.Child[I] END;

```

```

FUNCTION Parent_M4I( P : Pointer_M4I ) : Pointer_M4I;
BEGIN Parent_M4I := P^.Parent END;

PROCEDURE Ass_node_val_mst_M4I ( P : Pointer_M4I; I:INTEGER; Val : Typeelem_M4I);
BEGIN P^.Infor[I] := Val END;

PROCEDURE Ass_child_M4I( P : Pointer_M4I; I:INTEGER; Q : Pointer_M4I);
BEGIN P^.Child[I] := Q END;

PROCEDURE Aff_parent_M4I( P : Pointer_M4I; Q : Pointer_M4I);
BEGIN P^.Parent := Q END;

PROCEDURE Allocate_node_M4I( VAR P : Pointer_M4I ) ;
VAR
I : BYTE;
BEGIN
NEW ( P ) ;
For I:=1 TO 4 Do P^.Child[I] := NIL;
P^.degree := 0
END;

FUNCTION Degree_M4I ( P : Pointer_M4I ) : BYTE;
BEGIN
Degree_M4I := P^.Degree
END;

PROCEDURE Aff_Degree_M4I ( VAR P : Pointer_M4I; N : BYTE);
BEGIN
P^.Degree := N
END;

PROCEDURE Free_node_M4I( P : Pointer_M4I);
BEGIN
DISPOSE ( P )
END;

{ Declaration part of variables }
VAR
M : Pointeur_M4I;
{ Body of main program }
BEGIN
END.

```

Stack implementation in PASCAL

```

LET P : STACK ;
{ -Implementation- : STACK OF INTEGERS}

```

```

{ Stacks }
TYPE

```

```

Typeelem_PI = INTEGER; { Any type }
Pointer_PI = ^Maillon_PI ;
Maillon_PI = RECORD
Valeur : Typeelem_PI;
Next : Pointer_PI
END;

PROCEDURE Createstack_PI( VAR P : Pointer_PI );
BEGIN
P := NIL;
END;

FUNCTION Empty_stack_PI ( P : Pointer_PI ) : BOOLEAN;
BEGIN
Empty_stack_PI := ( P = NIL )
END;

PROCEDURE Push_PI ( VAR P : Pointer_PI; Val : Typeelem_PI );
VAR
Q : Pointer_PI;
BEGIN
NEW(Q);
Q^.Valeur := Val;
Q^.Next := P;
P := Q;
END;

PROCEDURE Pop_PI ( VAR P : Pointer_PI; VAR V :Typeelem_PI );
VAR Save : Pointer_PI;
BEGIN
IF NOT Empty_stack_PI (P)
THEN
BEGIN
V := P^.Valeur;
Save := P;
P := P^.Next;
DISPOSE(Save);
END
ELSE WRITELN('Pile Vide');
END;

{ Declaration part of variables }
VAR
P : Pointeur_PI;
{ Body of main program }
BEGIN
END.

```

Queue implementation in PASCAL

LET F : QUEUE ;


```

{ -Implementation- : QUEUE OF INTEGERS}

{ Queues }
TYPE
Typeelem_FI = INTEGER;
Ptliste_FI = ^Maillon_FI;
Maillon_FI = RECORD
Val : Typeelem_FI;
Next : Ptliste_FI
END;

Pointer_FI = ^ Filedattente_FI;
Filedattente_FI = RECORD
Tete, Queue : Ptliste_FI
END;

PROCEDURE Createqueue_FI (VAR Fil : Pointer_FI );
BEGIN
New (Fil);
Fil^.Tete := NIL ;
Fil^.Queue := Nil
END;

FUNCTION Empty_queue_FI (Fil : Pointer_FI) : BOOLEAN;
BEGIN Empty_queue_FI := Fil^.Tete = NIL END;

PROCEDURE Enqueue_FI (VAR Fil : Pointer_FI; Val : Typeelem_FI );
VAR
P : Ptliste_FI;
BEGIN
NEW(P);
P^.Val := Val;
P^.Next := NIL;
IF NOT Empty_queue_FI(Fil)
THEN Fil^.Queue^.Next := P
ELSE Fil^.Tete := P;
Fil^.Queue := P;
END;

PROCEDURE Dequeue_FI (VAR Fil : Pointer_FI ; VAR Val : Typeelem_FI );
BEGIN
IF NOT Empty_queue_FI(Fil)
THEN
BEGIN
Val := Fil^.Tete^.Val;
Fil^.Tete := Fil^.Tete^.Next;
END
ELSE WRITELN(' File Vide ');
END;

{ Declaration part of variables }

```

```

VAR
F : Pointeur_FI;
{ Body of main program }
BEGIN
END.

```

File implementation in PASCAL

```

LET F : FILE OF ( STRINGS , INTEGER ) HEADER ( INTEGER , INTEGER ) BUFFER B1 ;

```

```

{ -Implementation- : FILE }

```

```

{ Managing open files }

```

```

TYPE
_Ptr_Noead = ^_Noead;
_Noead = RECORD
Var_fich : Thandle;
Nom_fich : string;
Save_pos : Longint;
Next : _Ptr_Noead
END;

```

```

VAR
_stack_ouverts : _Ptr_Noead = NIL;

```

```

FUNCTION _Ouvert (Fp : String) : _Ptr_Noead;
VAR
P : _Ptr_Noead;
Found : boolean;
BEGIN
P := _Stack_Open; Found := False ;
WHILE (P <> NIL) AND NOT Found DO
IF P^.Nom_Fich = Fp
THEN Found := True
ELSE P := P^.Next;
 Ouvert := P;
END;

```

```

PROCEDURE _Push_ouvert ( Fp : string; VAR Fl: Thandle);
VAR
P : _Ptr_Noead ;
BEGIN
New(P);
P^.Nom_fich := Fp;
P^.Var_fich := Fl;
P^.Next := _Stack_Open;
_stack_Open := P
END ;

```

```

FUNCTION _Pop_ouvert ( Fl : Thandle) : String;

```

```

VAR
P, Prev : _Ptr_Noead ;
BEGIN
P:= _Stack_Open;
Prev := Nil;
WHILE P^.Var_fich <> FI DO
BEGIN Prev := P ; P := P^.Next END;
_Pop_ouvert := P^.Nom_fich ;
IF Prev <> NIL
THEN Prev^.Next := P^.Next
ELSE _Stack_Open := P^.Next;
Dispose (P);
END;

{ Files }
TYPE
{ Types of block fields }
Typefield1_SIEII = Typestring;
Typefield2_SIEII = INTEGER;

{ Type of file block structure }
Typestruct_SIEII = ^ Typestruct_SIEII_ ;
Typestruct_SIEII_ = RECORD
Field1 : Typefield1_SIEII ;
Field2 : Typefield2_SIEII ;
END;

{ Type of File data block }
Typestruct_SIEII_Buf = RECORD
Field1 : Typefield1_SIEII ;
Field2 : Typefield2_SIEII ;
END;

{ Types of header fields }
Typeentete1_SIEII = INTEGER;
Typeentete2_SIEII = INTEGER;

{ Type of file feature block }
Typestruct_SIEII_entete = RECORD
Entete1 : Typeentete1_SIEII ;
Entete2 : Typeentete2_SIEII ;
END;

{ Declaration of header block }

VAR
Buf_caract_SIEII : Typestruct_SIEII_entete ;

{ Operations on files }

```

```

PROCEDURE Open_SIEII (VAR Fl : Thandle ; Fp, Mode : STRING );
VAR
P : _Ptr_Noead;
BEGIN
P := _Open (Fp);
IF P <> NIL
THEN
BEGIN
{ Save the current position of the file and close it }
P^.Save_pos :=FILESEEK(P^.Var_fich, 0, 1);
FILESEEK(P^.Var_fich,0,0);
FILEWRITE(P^.Var_fich, Buf_caract_SIEII, sizeof(Buf_caract_SIEII) );
FILECLOSE (P^.Var_fich);
END;

{ Open or re open the file }
IF Mode = 'A'
THEN
BEGIN
Fl:=FILEOPEN(Fp,fmOpenReadWrite);
FILEREAD(Fl, Buf_caract_SIEII, sizeof(Buf_caract_SIEII) )
END
ELSE
BEGIN
Fl:=FILECREATE(Fp);
FILEWRITE(Fl, Buf_caract_SIEII, sizeof(Buf_caract_SIEII) )
END ;
_Push_Open(Fp, Fl);
END;

PROCEDURE Close_SIEII ( VAR Fl : Thandle);
VAR
P : _Ptr_Noead;
Fp : String;
BEGIN
Fp := _Pop_Open(Fl);

FILESEEK(Fl,0, 0);
FILEWRITE(Fl, Buf_caract_SIEII, sizeof(Buf_caract_SIEII) );
FILECLOSE(Fl);

{ Is there a file open with the same name? }
{ If yes, open it again at the saved position }
P := _Open (Fp);
IF P <> NIL
THEN
BEGIN
Fl:=FILEOPEN(P^.Nom_fich,fmOpenReadWrite);
FILEREAD(Fl, Buf_caract_SIEII, sizeof(Buf_caract_SIEII) );
FILESEEK(Fl, P^.Save_pos, 0)
END;

```

```

END;

FUNCTION Entete1_SIEII( VAR Fl : Thandle): Typeentete1_SIEII;
BEGIN
Entete1_SIEII := Buf_caract_SIEII.Entete1;
END;

FUNCTION Entete2_SIEII( VAR Fl : Thandle): Typeentete2_SIEII;
BEGIN
Entete2_SIEII := Buf_caract_SIEII.Entete2;
END;

PROCEDURE Aff_entete1_SIEII ( VAR Fl: Thandle; VAL : Typeentete1_SIEII);
BEGIN
Buf_caract_SIEII.Entete1 := VAL
END;

PROCEDURE Aff_entete2_SIEII ( VAR Fl: Thandle; VAL : Typeentete2_SIEII);
BEGIN
Buf_caract_SIEII.Entete2 := VAL
END;

PROCEDURE Writeseq_SIEII ( VAR Fl: Thandle; Buf : Typestruct_SIEII );
VAR
Buffer : Typestruct_SIEII_Buf ;
I : Integer;
BEGIN
Buffer.Field1:= Buf^.Field1;
Buffer.Field2:= Buf^.Field2;
FILEWRITE(Fl, Buffer, Sizeof(Buffer))
END;

PROCEDURE Writedir_SIEII ( VAR Fl: Thandle; Buf : Typestruct_SIEII; N: INTEGER );
VAR
Buffer : Typestruct_SIEII_Buf ;
I : Integer;
BEGIN
Buffer.Field1:= Buf^.Field1;
Buffer.Field2:= Buf^.Field2;
FILESEEK(Fl, Sizeof(Buf_caract_SIEII) + (N-1)*Sizeof(Buffer ), 0);
FILEWRITE(Fl, Buffer, Sizeof(Buffer))
END;

PROCEDURE Readseq_SIEII ( VAR Fl: Thandle; VAR Buf : Typestruct_SIEII );
VAR
Buffer : Typestruct_SIEII_Buf ;
I : Integer ;
BEGIN
FILEREAD(Fl, Buffer, Sizeof(Buffer));
Buf^.Field1:= Buffer.Field1;
Buf^.Field2:= Buffer.Field2;
END;

```

```

PROCEDURE Readdir_SIEII ( VAR Fl: Thandle; VAR Buf : Typestruct_SIEII; N: INTEGER );
VAR
Buffer : Typestruct_SIEII_Buf ;
I : Integer ;
BEGIN
FILESEEK(Fl, Sizeof(Buf_caract_SIEII) + (N-1)*Sizeof(Buffer ), 0);
FILEREAD(Fl, Buffer, Sizeof(Buffer));
Buf^.Field1:= Buffer.Field1;
Buf^.Field2:= Buffer.Field2;
END;

```

```

PROCEDURE Add_SIEII ( VAR Fl: Thandle; Buf : Typestruct_SIEII);
VAR
Buffer : Typestruct_SIEII_Buf ;
I : Integer;
BEGIN
Buffer.Field1:= Buf^.Field1;
Buffer.Field2:= Buf^.Field2;
FILESEEK(Fl, 0, 2);
FILEWRITE(Fl, Buffer, Sizeof(Buffer))
END;

```

```

FUNCTION Endfile_SIEII ( VAR Fl : Thandle): BOOLEAN;
VAR
K, K2 : Longint;
BEGIN
K := FILESEEK(Fl, 0, 1); { Current position }
K2 :=FILESEEK(Fl, 0, 2); { Last position }
IF K = K2
THEN
Endfile_SIEII := true
ELSE
BEGIN
FILESEEK(Fl, K, 0);
Endfile_SIEII := False
END;
END;

```

```

FUNCTION Alloc_block_SIEII ( VAR Fl : Thandle) : INTEGER;
VAR
K : Longint;
BEGIN
K := FILESEEK(Fl, 0, 2); { End of file }
K := K - Sizeof( Typestruct_SIEII_entete); { Ignore the header }
K := K DIV Sizeof (Typestruct_SIEII_Buf);
K := K + 1;
Alloc_block_SIEII := K;
END;

```

```

{ Declaration part of variables }
VAR

```

```
F : Thandle;  
B1 : Typestruct_SIEII ;  
{ Body of main program }  
BEGIN  
NEW(B1);  
END.
```

Translation from Z to C

Declarations
Expressions
Assignment
While loop
For loop
If statment
Reading
Writing
Action
Function
Standard functions
Algorithm

Variable declaration

A variable declaration in C is made by

 Type ;

where denotes an identifier list.

Simple objects

Equivalents $Z \rightarrow C$

INTEGER is translated to **int**.

CHAR is translated to **char**.

To type **STRING** we associate the built type String defined as follow:

```
typedef char Chaine[256]
```

Boolean type does not exist.

To continue working with this type, add at the beginning of the program

```
typedef int Boolean
```

Define also values True et False as follow :

```
#define true 1  
#define false 0
```

Structure objects

To define a structure S in C we must choose an implementation.

Implementing consists in choosing a memory representation (static or dynamic) and translate the operations of the abstract machine in this representation.

It suffices to replace the structure S by Pointer and add at the level of the header of the program C the desired implementation where the Pointer type will be defined.

Expressions

Z-expression grammar is included in C grammar.

Assignment

Same syntaxe with '=' instead of ':='.

While loop

```
-----Z-----  
WHILE Cond :  
    Statements  
ENDWHILE
```

is translated to :

```
-----C-----  
while ( Cond )  
{  
    Statements  
}
```

For loop

```
-----Z-----  
FOR V:= Exp1, Exp2 [, Exp3] [:]  
    Statements  
ENDFOR
```

is translated to :

```
-----C-----  
for (V=Exp1; V<=Exp2; V=V+Exp3)  
{  
    Statements  
}
```

If statement

```
-----Z-----  
IF Cond :  
    Statements  
[ELSE  
    Statements ]  
ENSIF
```

is translated to :

```
-----C-----  
if ( Cond )  
{  
    Statements  
}  
[else  
{Statements }]
```

Reading

-----Z-----
READ(V1, V2, ...)

is translated to :

-----C-----
scanf("...", &V1, &V2, ...)

Writing

-----Z-----
WRITE (E1, E2, ...)

is translated to :

-----C-----
printf(E1, E2, ...)

Actions

-----Z-----
ACTION Nom (P1, P2, ...);
LET
 Declaration of local objects and parameters
BEGIN
 Statements
END

is translated to :

-----C-----
void Nom (typ1 P1 , typ2 P2, ...)
{
 Declaration of local objects and parameters
 Statements
}

Functions

-----Z-----
FUNCTION Nom (P1, P2, ...) : type;
LET
 Declaration of local objects and parameters
BEGIN
 Statements
END

is translated to :

-----C-----
type Nom (typ1 P1, typ2 P2, ...)
{
 Declaration of local objects and parameters
 Statements
}

Standard functions

MOD (a, b)

```
int Mod( int a, int b)
{ return ( a % b ); }
```

MIN (a, b)

```
int Min (int a, int b)
{
    if (a < b) return(a);
    else return(b);
}
```

MAX (a, b)

```
int Max (int a, int b)
{
    if (a > b) return(a);
    else return(b);
}
```

EXP (a, b)

```
int Exp (int a, int b)
{
    int i; int Ex ;
    Ex = 1;
    for (i= 1; i<=b; i++)
        Ex = Ex * a ;
    return (Ex);
}
```

RANDNUMBER (N)

```
int Randnumber( int N )
{ return ( rand() % N ); }
```

RANDSTRING (N)

```
char *Randstring ( int N )
{
    int k;
    char * St = malloc(N);

    char Chr1[26] = "abcdefghijklmnopqrstuvwxyz";
    char Chr2[26] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    for (k=0;k<N; k++)
        switch ( rand() % 2 ){
            case 0 : *(St+k) = Chr1[rand() % 26] ; break ;
            case 1 : *(St+k) = Chr2[rand() % 26] ; break ;
        }
    St[k] = '\0' ;

    return (St);
}
```

```

STRINGLENGTH ( C )
int Stringlength ( string255 Ch )
{
    return strlen(Ch);
}

```

```

CHARACT(Ch, I)
char *Charact ( string255 Ch , int I )
{
    char *s = malloc(2);
    s[0] = Ch[I-1];
    s[1] = '\0';
    return s;
}

```

Algorithm

```

-----Z-----
LET
    Local and static objects
    Announcement of the modules
BEGIN
    Statements
END
Module 1
Module 2
...
Modules n

```

is translated to :

```

-----C-----
Local and static objects
// Definition of modules (Functions)
Module 1
Module 2
...
Module n

main()
{
    Statements
}

```

Implementation of Z machines in C

Arrays
Structures
Linked lists
Bidirectional linked lists
Stacks
Queues
Binary search trees
M-ary search trees
Files

Array implementation in C

```
LET Tab : ARRAY ( 5 , 10 );
```

```
/** -Implementation- **\: ARRAY OF INTEGERS**/
```

```
/** Arrays **/
```

```
typedef int Typeelem_V5_10i ;
```

```
typedef Typeelem_V5_10i * Typevect_V5_10i ;
```

```
Typeelem_V5_10i Element_V5_10i ( Typevect_V5_10i V , int I1 , int I2 )  
{  
    return *(V + (I1-1)*10 + (I2-1) ) ;  
}
```

```
void Ass_element_V5_10i ( Typevect_V5_10i V , int I1 , int I2, Typeelem_V5_10i Val )  
{  
    *(V + (I1-1)*10 + (I2-1) ) = Val ;  
}
```

```
/** Variables of main program **/
```

```
Typevect_V5_10i Tab;
```

```
/** Main program **/
```

```
int main(int argc, char *argv[])
```

```
{  
    Tab = malloc(5*10 * sizeof(int));  
}
```

Structure implementation in C

```
LET S : STRUCTURE ( STRING , INTEGER );
```

```
typedef char * string256 ;
```

```

/** Structures */
typedef struct Tsi Type_Tsi ;
typedef Type_Tsi * Typestr_Tsi ;
typedef string255 Type1_Tsi ;
typedef int Type2_Tsi ;
struct Tsi
{
Type1_Tsi Field1 ;
Type2_Tsi Field2 ;
};

Type1_Tsi Struct1_Tsi ( Typestr_Tsi S)
{
return S->Field1 ;
}

Type2_Tsi Struct2_Tsi ( Typestr_Tsi S)
{
return S->Field2 ;
}

void Ass_struct1_Tsi ( Typestr_Tsi S, Type1_Tsi Val )
{
strcpy( S->Field1 , Val );
}

void Ass_struct2_Tsi ( Typestr_Tsi S, Type2_Tsi Val )
{
S->Field2 = Val ;
}

/** Variables of main program */
Typevect_V5_10i Tab;
Typestr_Tsi S;
/** Main program */
int main(int argc, char *argv[])
{
S = malloc(sizeof(Typestr_Tsi));
S->Field1 = malloc(sizeof(string256));
}

```

Linked list implementation in C

LET L : LIST ;

/** -Implementation- */: LIST Of INTEGERS**/

/** Linked lists */

```

typedef int Typeelem_Li ;
typedef struct Maillon_Li * Pointer_Li ;

```

```

struct Maillon_Li
{
Typeelem_Li Val ;
Pointer_Li Next ;
} ;

Pointer_Li Allocate_cell_Li (Pointer_Li *P)
{
*P = (struct Maillon_Li *) malloc( sizeof( struct Maillon_Li)) ;
(*P)->Next = NULL;
}

void Ass_val_Li(Pointer_Li P, Typeelem_Li Val)
{
P->Val = Val ;
}

void Ass_adr_Li( Pointer_Li P, Pointer_Li Q)
{
P->Next = Q ;
}

Pointer_Li Next_Li( Pointer_Li P)
{ return( P->Next ) ; }

Typeelem_Li Cell_value_Li( Pointer_Li P)
{ return( P->Val ) ; }

void Free_Li ( Pointer_Li P)
{ free (P);}

/** Variables of main program **/
Pointeur_Li L=NULL;
/** Main program **/
int main(int argc, char *argv[])
{
}

Bidirectional linked list implementation in C
LET L : BILIST ;

/** -Implementation- **/: BIDIRECTIONAL LIST OF INTEGERS**/

/** Bidirectional linked lists **/

typedef int Typeelem_Ri ;
typedef struct Maillon_Ri * Pointer_Ri ;

struct Maillon_Ri
{
Typeelem_Ri Val ;
Pointer_Ri Next ;

```

```

Pointer_Ri Prev ;
} ;

Pointer_Ri Allocate_cell_Ri (Pointer_Ri *P)
{
    *P = (struct Maillon_Ri *) malloc( sizeof( struct Maillon_Ri)) ;
    (*P)->Next = NULL;
    (*P)->Prev = NULL;
}

void Ass_val_Ri(Pointer_Ri P, Typeelem_Ri Val)
{
    P->Val = Val ;
}

void Ass_r_adr_Ri( Pointer_Ri P, Pointer_Ri Q)
{ P->Next = Q; }

void Ass_l_adr_Ri( Pointer_Ri P, Pointer_Ri Q)
{ P->Prev = Q; }

Pointer_Ri Next_Ri( Pointer_Ri P)
{ return( P->Next); }

Pointer_Ri Previous_Ri( Pointer_Ri P)
{ return( P->Prev); }

Typeelem_Ri Cell_value_Ri( Pointer_Ri P)
{ return( P->Val); }

void Free_Ri ( Pointer_Ri P)
{ free (P) ; }

/** Variables of main program **/
Pointeur_Ri L=NULL;

/** Main program **/
int main(int argc, char *argv[])
{
}

```

Binary search tree implementation in C

LET A : BST ;

```

/** -Implementation- **\: BINARY SERACH TREE OF INTEGERS**/

/** Binary search trees **/

typedef int Typeelem_Ai ;
typedef struct Noeud_Ai * Pointer_Ai ;

```



```

struct Noeud_Ai
{
Typeelem_Ai Val ;
Pointer_Ai Lc ;
Pointer_Ai Rc ;
Pointer_Ai Parent ;
} ;

Typeelem_Ai Node_value_Ai( Pointer_Ai P )
{ return P->Val; }

Pointer_Ai Lc_Ai( Pointer_Ai P)
{ return P->Lc ; }

Pointer_Ai Rc_Ai( Pointer_Ai P)
{ return P->Rc ; }

Pointer_Ai Parent_Ai( Pointer_Ai P)
{ return P->Parent ; }

void Ass_node_val_Ai ( Pointer_Ai P, Typeelem_Ai Val)
{
P->Val = Val ;
}

void Ass_lc_Ai( Pointer_Ai P, Pointer_Ai Q)
{ P->Lc = Q; }

void Ass_rc_Ai( Pointer_Ai P, Pointer_Ai Q)
{ P->Rc = Q ; }

void Ass_parent_Ai( Pointer_Ai P, Pointer_Ai Q)
{ P->Parent = Q ; }

void Allocate_node_Ai( Pointer_Ai *P)
{
*P = (struct Noeud_Ai *) malloc( sizeof( struct Noeud_Ai)) ;
(*P)->Lc = NULL;
(*P)->Rc = NULL;
(*P)->Parent = NULL;
}

void Free_node_Ai( Pointer_Ai P)
{ free( P ) ; }

/** Variables of main program */
Pointeur_Ai A=NULL;
/** Main program */
int main(int argc, char *argv[])
{
}

```

M-ary search tree implémentation in C

LET M : MST (4) ;

```
/** -Implementation- **\: M-ARY SEARCH TREE OF INTEGERS**/
```

```
/** M-ary search trees **/
```

```
typedef int Typeelem_M4i ;  
typedef struct Noeud_M4i * Pointer_M4i ;
```

```
struct Noeud_M4i  
{  
    int Degree ;  
    Pointer_M4i Child[4] ;  
    Typeelem_M4i Infor[4] ;  
    Pointer_M4i Parent ;  
} ;
```

```
Typeelem_M4i Node_value_mst_M4i(Pointer_M4i P, int I)  
{ return P->Infor[I-1] ; }
```

```
Pointer_M4i Child_M4i( Pointer_M4i P, int I)  
{ return P->Child[I-1] ; }
```

```
Pointer_M4i Parent_M4i( Pointer_M4i P)  
{ return P->Parent ; }
```

```
void Ass_node_val_mst_M4i ( Pointer_M4i P, int I, Typeelem_M4i Val)  
{  
    P->Infor[I-1] = Val ;  
}
```

```
void Ass_child_M4i( Pointer_M4i P, int I, Pointer_M4i Q)  
{ P->Child[I-1] = Q ; }
```

```
void Aff_parent_M4i( Pointer_M4i P, Pointer_M4i Q)  
{ P->Parent = Q ; }
```

```
void Allocate_node_M4i( Pointer_M4i *P )  
{  
    int I ;
```

```
    *P = (struct Noeud_M4i *) malloc( sizeof( struct Noeud_M4i)) ;  
    for (I=0; I< 4; ++I) (*P)->Child[I] = NULL;  
    (*P)->Degree = 0 ;  
}
```

```
int Degree_M4i ( Pointer_M4i P )  
{ return P->Degree ; }
```

```
void Ass_degree_M4i ( Pointer_M4i P, int N)  
{ P->Degree = N ; }
```

```
void Free_node_M4i(Pointer_M4i P)
{ free ( P );}
```

```
/** Variables of main program */
Pointeur_M4i M=NULL;
/** Main program */
int main(int argc, char *argv[])
{
}
```

Stack implementation in C

LET P : STACK ;

```
/** -Implementation- **\: STACK OF INTEGERS**/
/** Stacks**/
```

```
typedef int Typeelem_Pi ;
typedef struct Maillon_Pi * Pointer_Pi ;
typedef Pointer_Pi Typepile_Pi ;
```

```
struct Maillon_Pi
{
Typeelem_Pi Val ;
Pointer_Pi Next ;
} ;
```

```
void Createstack_Pi( Pointer_Pi *P )
{ *P = NULL ; }
```

```
bool Empty_stack_Pi ( Pointer_Pi P )
{ return (P == NULL ); }
```

```
void Push_Pi ( Pointer_Pi *P, Typeelem_Pi Val )
{
Pointer_Pi Q;
```

```
Q = (struct Maillon_Pi *) malloc( sizeof( struct Maillon_Pi) ) ;
Q->Val = Val ;
Q->Next = *P;
*P = Q;
}
```

```
void Pop_Pi ( Pointer_Pi *P, Typeelem_Pi *Val )
{
Pointer_Pi Save;
```

```
if (! Empty_stack_Pi (*P) )
{
*Val = (*P)->Val ;
Save = *P;
```

```

    *P = (*P)->Next;
    free(Save);
}
else printf ("%s \n", "Stack is empty");
}

```

```

/** Variables of main program */
Pointeur_Pi P=NULL;
/** Main program */
int main(int argc, char *argv[])
{
}

```

Queue implementation in C

LET F : QUEUE ;

```

/** -Implementation- **\: QUEUE OF INTEGERS**/
/** Queues **/

```

```

typedef int Typeelem_Fi ;
typedef struct Filedattente_Fi * Pointer_Fi ;
typedef struct Maillon_Fi * Ptliste_Fi ;

```

```

struct Maillon_Fi
{
    Typeelem_Fi Val ;
    Ptliste_Fi Next ;
};

```

```

struct Filedattente_Fi
{
    Ptliste_Fi Head, Queue ;
};

```

```

void Createqueue_Fi ( Pointer_Fi *Fil )
{
    *Fil = (struct Filedattente_Fi *) malloc( sizeof( struct Filedattente_Fi) ) ;
    (*Fil)->Head = NULL ;
    (*Fil)->Queue = NULL ;
}

```

```

bool Empty_queue_Fi (Pointer_Fi Fil )
{ return Fil->Head == NULL ;}

```

```

void Enqueue_Fi ( Pointer_Fi Fil , Typeelem_Fi Val )
{
    Ptliste_Fi Q;

```

```

    Q = (struct Maillon_Fi *) malloc( sizeof( struct Maillon_Fi) ) ;
    Q->Val = Val ;

```

```

Q->Next = NULL;
if ( ! Empty_queue_Fi(Fil) )
Fil->Queue->Next = Q ;
else Fil->Head = Q;
Fil->Queue = Q;
}

void Dequeue_Fi (Pointer_Fi Fil, Typeelem_Fi *Val )
{
if (! Empty_queue_Fi(Fil) )
{
*Val = Fil->Head->Val ;
Fil->Head = Fil->Head->Next;
}
else printf ("%s \n", "Queue is empty");
}

/** Variables of main program **/
Pointeur_Fi F=NULL;
/** Main program **/
int main(int argc, char *argv[])
{
}

```

File implementation in C

LET F : FILE OF (STRINGS , INTEGER) HEADER (INTEGER , INTEGER) BUFFER B1 ;

```

/** -Implementation- **\: FILE **/

/* Managing open files */

struct _Node
{
FILE * Var_file ;
char * Name_file ;
int Save_pos;
struct _Node *Suiv ;
} ;

typedef struct _Node * _Ptr_Node;

_Ptr_Node _Stack_Opens = NULL;

/* Test if a file is open */
_Ptr_Node _Open ( char * Fp)
{
_Ptr_Node P;
bool Trouv ;
P = _Stack_Opens; Trouv = False ;
while ((P != NULL) && ! Trouv )
if ( strcmp(P->Name_file, Fp) == 0)

```

```

Trouv = True;
else P = P->Suiv;
return P;
}

/* Add an open file */
void _Push_Open ( char *Fp, FILE *Fl)
{
    _Ptr_Node P ;
    P = (_Ptr_Node) malloc( sizeof( struct _Node)) ;
    P->Name_file = Fp;
    P->Var_file = Fl;
    P->Suiv = _Stack_Opens;
    _Stack_Opens = P;
}

/* Delete an open file and return its name */
char * _Pop_Open ( FILE *Fl)
{
    char * Fp = malloc (100);
    _Ptr_Node P, Prec ;
    P = _Stack_Opens;
    Prec = NULL;
    while (P->Var_file != Fl )
    { Prec = P ; P = P->Suiv ;}
    strcpy(Fp, P->Name_file);
    if (Prec != NULL)
        Prec->Suiv = P->Suiv;
    else _Stack_Opens = P->Suiv;
    free (P);
    return Fp ;
}

/** Files **/

typedef char _Tx[255];
/** Types of block fields **/
typedef string255 Typefield1_siEii;
typedef _Tx Typefield1_siEii_Buf ;
typedef int Typefield2_siEii;

/** Types of header fields **/
typedef int Typeentete1_siEii ;
typedef int Typeentete2_siEii ;

/** Type of file data block **/
typedef struct
{
    Typefield1_siEii_Buf Field1 ;
    Typefield2_siEii Field2 ;
} Typestruct1_siEii_Buf;

```

```

/** Type of File data block structure */
typedef struct
{
Typefield1_siEii Field1 ;
Typefield2_siEii Field2 ;
} Typestruct1_siEii ;
typedef Typestruct1_siEii_ * Typestruct1_siEii ;

/** Type of block of file features */
typedef struct
{
Typeentete1_siEii Entete1 ;
Typeentete2_siEii Entete2 ;
} Typestruct2_siEii ;

/** Declaration of header block */

Typestruct2_siEii Bloc_caract_siEii;

/** Operations on files */

void Open_siEii ( FILE **siEii , char *Fp , char * Mode )
{
_Ptr_Node P = _Open(Fp);
if ( P != NULL )
/* The file is already open */
{
P->Save_pos = ftell (P->Var_file);
fseek( P->Var_file, 0, 0);
fwrite(&Bloc_caract_siEii, sizeof(Typestruct2_siEii), 1, P->Var_file);
fclose(P->Var_file);
}
/* The file is not open */
if ( strcmp(Mode,"A") == 0)
{
*siEii = fopen(Fp, "r+b");
fread(&Bloc_caract_siEii, sizeof(Typestruct2_siEii), 1, *siEii);
}
else
{
*siEii = fopen(Fp, "w+b");
fwrite(&Bloc_caract_siEii, sizeof(Typestruct2_siEii), 1, *siEii) ;
}
_Push_Open( Fp, *siEii);
}

void Close_siEii ( FILE * siEii )
{
char * Fp = malloc(100);
_Ptr_Node P ;
strcpy(Fp, _Pop_Open(siEii));

```

```

fseek( siEii, 0, 0);
fwrite(&Bloc_caract_siEii, sizeof(Typestruct2_siEii), 1, siEii);
fclose(siEii) ;
/* Is there a file open with the same name? */
/* If yes, open it again at the saved position */
P = _Open (Fp);
if ( P != NULL)
{
    siEii = fopen(P->Name_file, "r+b");
    fread(&Bloc_caract_siEii, sizeof(Typestruct2_siEii), 1, siEii);
    fseek(siEii, P->Save_pos, 0);
}
}

```

```

Typeentete1_siEii Entete1_siEii( FILE * siEii)
{
    return Bloc_caract_siEii.Entete1;
}

```

```

Typeentete2_siEii Entete2_siEii( FILE * siEii)
{
    return Bloc_caract_siEii.Entete2;
}

```

```

void Aff_entete1_siEii ( FILE * siEii, Typeentete1_siEii VAL)
{
    Bloc_caract_siEii.Entete1 = VAL ;
}

```

```

void Aff_entete2_siEii ( FILE * siEii, Typeentete2_siEii VAL)
{
    Bloc_caract_siEii.Entete2 = VAL ;
}

```

```

void Writeseq_siEii ( FILE * siEii, Typestruct1_siEii Buf )
{
    Typestruct1_siEii_Buf Buffer ;
    int I, J;
    for(J=0; J<= strlen(Buf->Field1); ++J)
        Buffer.Field1[J] = Buf->Field1[J];
    Buffer.Field2 = Buf->Field2;
    fwrite(&Buffer, sizeof( Typestruct1_siEii_Buf), 1, siEii) ;
}

```

```

void Writedir_siEii ( FILE * siEii, Typestruct1_siEii Buf, int N )
{
    Typestruct1_siEii_Buf Buffer ;
    int I, J;
    for(J=0; J<= strlen(Buf->Field1); ++J)
        Buffer.Field1[J] = Buf->Field1[J];
    Buffer.Field2 = Buf->Field2;
    fseek(siEii, (long) ((N-1)* sizeof( Typestruct1_siEii_Buf) +

```



```

sizeof( Typestruct2_siEii)), 0 );
fwrite(&Buffer, sizeof( Typestruct1_siEii_Buf), 1, siEii );
}

void Readseq_siEii ( FILE * siEii, Typestruct1_siEii Buf )
{
Typestruct1_siEii_Buf Buffer ;
int I, J;
if (fread(&Buffer, sizeof( Typestruct1_siEii_Buf), 1, siEii)!=0){
for(J=0; J<= strlen(Buffer.Field1); ++J)
Buf->Field1[J] = Buffer.Field1[J];
Buf->Field2= Buffer.Field2;
}
}

void Readdir_siEii ( FILE * siEii, Typestruct1_siEii Buf, int N)
{
Typestruct1_siEii_Buf Buffer ;
int I, J;
fseek(siEii, (long) ((N-1)* sizeof( Typestruct1_siEii_Buf) +
sizeof( Typestruct2_siEii)), 0 );
fread(&Buffer, sizeof( Typestruct1_siEii_Buf), 1, siEii);
for(J=0; J<= strlen(Buffer.Field1); ++J)
Buf->Field1[J] = Buffer.Field1[J];
Buf->Field2= Buffer.Field2;
}

void Add_siEii ( FILE * siEii, Typestruct1_siEii Buf )
{
Typestruct1_siEii_Buf Buffer ;
int I, J;
for(J=0; J<= strlen(Buf->Field1); ++J)
Buffer.Field1[J] = Buf->Field1[J];
Buffer.Field2 = Buf->Field2;
fseek(siEii, 0, 2); /* End of file */
fwrite(&Buffer, sizeof( Typestruct1_siEii_Buf), 1, siEii );
}

bool Endfile_siEii (FILE * siEii)
{
long K = ftell(siEii);
fseek(siEii, 0, 2); /* End of file */
long K2 = ftell(siEii); /* Position from beginning */
if (K==K2)
{ fseek(siEii, K, 0); return 1;}
else
{ fseek(siEii, K, 0); return 0;}
}

int Alloc_block_siEii (FILE * siEii)
{
long K;

```

```

fseek(siEii, 0, 2); /* End of file */
K = ftell(siEii); /* Position from beginning */
K = K - sizeof( Typestruct2_siEii); /* Ignore the header */
K = K / sizeof (Typestruct1_siEii_Buf);
K ++;
return(K);
}

/** Variables of main program */
FILE *F;
Typestruct1_siEii B1 ;
/** Main program */
int main(int argc, char *argv[])
{
B1 = malloc(sizeof(Typestruct1_siEii));
B1->Field1 = malloc(255 * sizeof(string255));
}

```

Keyword Index

A

ACTION	ACTIONS
ADD	
AND	
ARRAY	ARRAYS
ASS_ADR	
ASS_R_ADR	
ASS_L_ADR	
ASS_DEGREE	
ASS_ELEMENT	
ASS_HEADER	
ASS_RC	
ASS_LC	
ASS_CHILD	
ASS_NODE_VAL	
ASS_NODE_VAL_MST	
ASS_PARENT	
ASS_STRUCT	
ASS_VAL	
ALLOCATE_NODE	
ALLOC_BLOCK	
ALLOC_STRUCT	
ALLOC_ARRAY	
ALLOCATE	

B

BEGIN
BILIST
BOOLEAN
BOOLEANS
BST
BUFFER

C

CALL		
CHAR	CHARACTER	CHARACTERS
CHILD		
CLOSE		
CREATEQUEUE		
CREATESTACK		
CREATE_BST		
CREATE_MST		
CREATE_QUEUE		

CREATE_LIST
CREATE_BILIST
CREATE_STACK

D

DEGREE
DEQUEUE
DYNAMIC

E

EMPTY_STACK
EMPTY_QUEUE
END
ENDFILE
ENDFOR
EFOR
ENDIF
ENDWHILE
ELEMENT
ELSE
ENQUEUE
EWH

F

FALSE
FILE FILES
FOR
FREE
FREE_STRUCT
FREE_ARRAY
FREENODE
FUNCTION FUNCTIONS

H

HEADER

I

IF
INIT_STRUCT
INIT_ARRAY INIT_VECT
INTEGER INTEGERS

L

LC
LET
LIST LISTS

M

MAX
MIN
MOD
MST

N

NEXT
NIL
NODE_VALUE
NODE_VALUE_MST
NOT

O

OF
OR
OPEN

P

PARENT
POP
POINTER POINTERS
PREVIOUS
PUSH

Q

QUEUE QUEUES

R

RC
READ
READDIR
READSEQ

S

STACK STACKS
STRING STRINGS
STRUCT
STRUCTURE STRUCTURES

T

To

V

VALUE

W

WHILE

WRITE

WRITEDIR

WRITESEQ

WH