The Programming Language Io

Massimo Ancona and **Walter Cazzola**

DISI-Department of Informatics and Computer Science
University of Genova, Italy
{ancona|cazzola}@disi.unige.it

1 Introduction

Io is an experimental programming language designed for *reflective programming*. Io is inspired by Oberon, Modula-2, Pascal and C. Io has been designed as a tool for understanding the essence of *reflection* and for teaching *reflective programming* and its implementation mechanisms in under-graduate courses, thus Io is a *very concise language*. Nevertheless, Io is not a toy language: it is located at a mid distance between toys languages and production languages - despite its simplicity, it can be used for developing complex and modular programs that can be executed with a good efficiency. Io has been kept as simple as possible by discarding all those features neither implicitly nor explicitly connected with reflective programming. Reflection is based on the following concept:

> "... a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) *formally manipulating representations of its own operations and structures."* [2]

From the above definition, we can observe that:

- Reflection is not tight to specific linguistic constructs, but mainly a property of the run-time execution environment RTE[1] and of specific mechanisms entrusted into its run-time system library RTS.
- Moreover, the role of an interpreter is not essential: the same mechanisms can be more efficiently (and easily) implemented into a compiled language.

In other words *any reasonable programming language* can be made reflective *without consistent extensions* by enhancing its RTE/RTS with adequate capabilities and by adding only few predefined procedures. This fact does not preclude from adding to the language specific constructs for entrusting into it reflective features. What we assert here is that conceptually reflection is a very low level paradigm not necessarily tight to specific syntactic and semantic structures of a language, even if they may enhance program readability and development (see, for example the well-known meta-object protocol [1]).

It follows that, if we restrict our attention only to the language constructs we cannot consider Io as an innovative language: its originality lies in the architecture of its RTE,

[1] We define as RTE the set of machine-code (virtual intermediate code for interpreters) and data structures of the translated program together with all data structures and dedicated machine (or virtual) register, plus the run-time system library (RTS) supporting a program execution.

that makes it strongly tight to reflection from its early design phase. By the way, the main features of the language are:

- Modularity, similar to that of Modula-2, that supports modular programming via the mechanism of separate compilations enhanced with *module inheritance*.
- Support for complex data types and data abstraction for the development of abstract data-types.
- Simple implementation — the size of the compiler source code (written in Io) is about 2400 statements, that of the interpreter (written in Pascal and C) 500-700 statements.

This report is a concise reference for people interested in exploiting Io's reflective capabilities. Its function is to serve as a quick reference for supporting the development of reflective programs and for understanding how reflection is entrusted into the RTE of Io.

To this purpose, Io has been completely implemented via an interpreted virtual machine called `IoVM`. However, we stress that our concept of reflection does not require an interpreted virtual machine approach like other reflective systems — our method can be more easily implemented for compiled programming languages (e.g. Oberon, Modula-2, C++ or C) with a remarkable gain in efficiency.

The name Io has two motivations: it comes from the Italian pronoun "io" ("the self" or "I myself") which is significantly "reflective", and from the name of a satellite of Jupiter, discovered by Galileo on 1610, which is a tribute to the programming language Oberon[2], which in turn borrows its name from a satellite of Neptune.

2 Syntax

The meta-language used in this report to specify the syntax of Io is based on the extended Backus-Naur formalism (EBNF). Brackets '[' and ']' denote optionality of the enclosed sentential form, and braces '{' and '}' denote its repetition (possibly 0 times). Syntactic entities (non-terminal symbols) are denoted by English words expressing their intuitive meaning. Symbols of the language vocabulary (lexical tokens or terminal symbols) are denoted by strings enclosed in quote marks or words written in capital letters, the so-called *reserved words*. Each EBNF rule is terminated by a dot.

3 Lexical Tokens

The lexical tokens used to construct Io programs are sequences of ASCII characters. Tokens are classified into: identifiers, keywords, numbers, strings, operators, delimiters, and comments. Blanks and line breaks must not occur within symbols (except in comments, and blanks in strings); they are ignored unless they are essential to separate two consecutive symbols. Io is case sensitive: capital and lower-case letters are considered as being distinct.

[2] And to its precursors Modula-2 and Pascal that strongly influenced Io architecture.

3.1 Identifiers

Identifiers are sequences of letters, underscore character and digits. The first character cannot be a digit. Identifiers are symbols used to denote constants, types, variables, procedures, modules and record fields.

```
IdChar = Letter | "_".
Ident = IdChar { IdChar | Digit }.
```

Examples:

```
x   scan  _Io_ Eos   Get_Symbol   Number
```

3.1.1 Reserved Identifier (or Keywords)

The following identifiers are reserved for use as keywords, they cannot be used otherwise:

```
AND    ARRAY     AS       BEGIN   CASE       CONST    CYCLE    DIV
DO     DOWNTO    ELSE     ELSIF   END        EXIT     FI       FOR
FO     FUNCTION  GLOBAL   IF      IN         MOD      MODULE   NIL
NOT    OD        OF       OR      PROCEDURE  RECORD   RETURN   SET
THEN   TO        TYPE     UNTIL   USE        VAR      WHILE
```

3.2 Numbers

Numbers are (unsigned) integers or real numbers. Integers are sequences of digits and may be followed by a suffix letter. If no suffix is specified, the representation is decimal. The suffix `H` indicates hexadecimal representation.

A real number always contains a decimal point. Optionally it may also contain a decimal scale factor that is preceded by the letter `E`, that means "times ten to the power of." A real number is of type `REAL`.

```
Number = Integer | Real.
Integer = Digit { Digit } | Digit { HexDigit } "H".
Real = Digit { Digit } "." { Digit } [ ScaleFactor ].
ScaleFactor = "E" [ "+" | "-" ] Digit { Digit }.
HexDigit = Digit | "A" | "B" | "C" | "D" | "E" | "F".
Digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

Examples:

```
1997    100H    12.3    4.567E8    0.57712566E-6
```

3.3 Character Literals

Character literals are either a single ASCII character enclosed in quote marks or by the ordinal number of the character in hexadecimal notation followed by the letter `X`. The character ' is written ''.

```
CharConstant = "'" char "'" | Digit { HexDigit } "X".
```

Examples:

```
Hex_Int = 01FFFH; CR = 0DX; LF = 0AX;  U  = 'U'; QM = '''';
```

3.4 String Literal

A *string literal* is a character sequence enclosed in quote marks (’). Any quote mark in it has to be doubled. The number of characters in a string literal is called the *length* of the string. The type of a string literal is `ARRAY[1..n] OF CHAR`, where `n` is the number of character in the string. A string can be assigned to and compared with arrays of characters of the same type and of the same length.

```
String = ’ { character } ’.
```

Examples:

```
’Io’    ’Don’’t worry!’
```

3.5 Set Literal

A *set literal* is a value of type set whose syntax is the following:

```
Set = "[" [ Element { "," Element } ] "]".
Element = OrdinalConstant [ ".." OrdinalConstant].
OrdinalConstant= Char | Integer.
```

where ”Constant” above denotes literals of the same ordinal type.

3.6 Special Symbols

Special Symbols are operators and delimiters used to delimit the syntactic units of Io. They are listed below:

```
+   *   =   ,   #    ;   |   -    /    (    )    [   ]   &
\   ~   ^   .   ..   >   <   >=   <=   <>   :=   ’   :
```

3.7 Comments

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

4 Scope and Declarations

An identifier is a name denoting a data entry and must be introduced by a declaration, unless it is a predefined identifier. Declarations also serve to specify certain permanent properties of an object, such as whether it is a constant, a type, a module, a variable, or a subprogram. The name is used to refer to associated entity and can only be used within a region of the program called its *scope.* Each declaration is a definition of the declared identifier, unless the identifier is specified into a *use clause* or is the name of a subprogram declaration which does not provides a body (*forward declaration*).

No identifier may denote more than one object within a given scope. The scope extends textually from the point of the declaration to the end of the block (procedure or

module) to which the declaration belongs and hence to which the object is *local*. The scope rule has the following exceptions:

❶ If a type `T` is defined as `^T1` (see §6.5), the identifier `T1` can be declared textually following the declaration of `T`, but it must lie within the same scope.

❷ Field identifiers of a record declaration (see §6.4) are valid in field designators only.

A *top-level entity* (either variable, type, constant or subprogram) is an entity declared in the outermost scope of a module, any other entity is a *local* entity. Every identifier in the global scope (top-level) may be included into an export list (`GLOBAL`)[3] to indicate that it is exported from its declaring module. In this case, the identifier is accessible from other modules that specify it, or the enclosing module, in a *use list*. In the first case the identifier may be freely used in the importing module, in the second case the identifier may be referred in qualified notation, i.e. by prefixing it by the name of the declaring module followed a period (see section 11). The prefix and the identifier (with the separating period) are called a *qualified identifier*.

```
QualIdent = [ Ident "." ] Ident.
IdentDef = Ident [ "*" ].
ExportClause = "GLOBAL" IdList ";"
```

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

ABS	(9.2)	EXP	(9.2)	NIL	(6.5)	ROUND	(9.2)
ARCTAN	(9.2)	FALSE	(6.1)	ODD	(9.2)	SIN	(9.2)
BOOLEAN	(6.1)	FREE	(6.5)	ORD	(9.2)	SQR	(9.2)
CHAR	(6.1)	HALT	(9.2)	OUTPUT	(10)	SQRT	(9.2)
CHR	(9.2)	IDLE	(9.2)	PRED	(6.1)	SUCC	(6.1)
CLOSE	(10)	INT	(9.2)	READ	(10)	TEXT	(6.1)
COS	(9.2)	INPUT	(10)	READLN	(10)	TRUE	(6.1)
EMIT	(??)	INTEGER	(6.1)	REAL	(6.1)	TRUNC	(9.2)
EOF	(10)	LN	(9.2)	RESET	(10)	WRITE	(10)
EOLN	(10)	NEW	(6.5)	REWRITE	(10)	WRITELN	(10)

5 Constant Declarations

A constant declaration is used to give a name to a constant value. A constant value is a value that does not change during the program execution.

```
ConstantDeclaration = IdentDef "=" ConstantValue.
ConstantValue = Number | CharConstant | String | ConstantName.
```

Examples of constant declarations are

```
N = 100
null = 0X0
FF   = 0XFF
all = 'xyzw'
m   = N
```

[3] Record fields belonging to exported records are considered all visible in the present implementation.

6 Types and Type Declarations

A data type specifies the set of values which a given data item may have, and the operations that could be performed upon it. A type definition introduces an identifier to denote a type. In lo two data items have the same type, if the respective declarations the same type identifier is used, for specifying their type, or both are declared in the same declaration by means of the same *unnamed* type definition. In other words, lo uses a strict *name equivalence* rule for types. Types in lo may be basic types, or structured types. Basic types are `INTEGER`, `BOOLEAN`, `CHAR` and `REAL` and are predefined. There are three different structures, namely arrays, records and sets, with different component selectors.

```
TypeDeclaration = IdentDef "=" Type.
Type =  ArrayType | RecordType | PointerType | SetType | SubprogType.
```

Examples:

```
CONST c1=10; tmax=1000;
TYPE alfa = ARRAY[1..c1] OF CHAR;
  wordref = ^word;
  itemref = ^item;
  word = RECORD
           key:alfa;
           first, last:itemref;
           left, right:wordref
         END;
  item = RECORD
           ln0: INTEGER;
           next: itemref
         END;
  symset = SET OF 0..32;
```

6.1 Basic Types

lo provides the following basic types (denoted by predeclared identifiers) together with a set of predefined function and procedures (described in 8.2 and 10.2) operating upon them. The values of a given basic type are the following:

1. `BOOLEAN` the truth values `TRUE` and `FALSE`.
2. `CHAR` the characters of the extended ASCII set `(0X..0FFX)`.
3. `INTEGER` the integers between *minint* and *maxint* (hardware-dependent).
4. `REAL` real numbers between *minreal* and *maxreal* (hardware-dependent).
5. `TEXT` text-streams: textual sequential files.

The type `INTEGER` is included in type `REAL`, in other words:

$$\texttt{REAL} \supseteq \texttt{INTEGER}$$

The above relation causes coercion of an integer value to the corresponding real value whenever a real value is expected and an integer is provided instead. Types 1, 2 and 3

together are called *ordinal* types. The operators SUCC and PRED are defined for ordinal values and respectively compute the successor and the predecessor of their arguments when defined.

6.2 Array Types

An array is an indexed structure consisting of a number of components, all of the same type called the *element type*. The individual elements are designated by an index with values in an *range* [a..b] of any ordinal type, called the *index type* of the array. The number of elements (b-a+1) of an array is called its *length*.

```
ArrayType = ARRAY "[" Index { "," Index }"]" OF Type.
Index = OrdinalConstant ".." OrdinalConstant.
```

A declaration of the form:

```
ARRAY [L0..N0, L1..N1, ..., Lk..Nk] OF T
```

is a shorthand of the declaration

```
ARRAY [L0..N0] OF ARRAY [L1..N1] OF ... ARRAY [Lk..Nk] OF T
```

Examples of array types:

```
ARRAY [0,N] OF INTEGER
ARRAY [1..10,-20..20] OF REAL
```

6.2.1 Alpha Strings

There is a useful kind of array whose components are characters and index type is an interval of integers [1..n]. This kind of array is called an *alpha-string*. Examples:

```
TYPE string=ARRAY[1..6] OF CHAR;
VAR s:string;
  s:='abcdef'; (* OK *)
  s:='abcdefgh'; (* error different string lengths *)
```

6.3 Set Types

A *set* is a collection of values called elements all belonging to the same ordinal type called the *element type.*

```
SetType = "SET OF" OrdinalType.
OrdinalType = INTEGER | CHAR.
```

where Base is an interval of ordinal type. Examples

```
TYPE S = SET OF 1..10; CharSet = SET OF 0X..0FFX;
```

lo supports literal of type SET *a la* Pascal. For example, with the above types one can write:

```
VAR s:S; s1: CharSet;
  s := [1,3,5-8,10]; s1 := ['a'..'z'];
```

6.4 Record Types

A record type is a sequence of components, called *fields* each with its own type, and name (called *field identifier*). The scope of these field identifiers is the record definition itself, but they are also visible within field designators (see 8.2) referring to elements of record variables.

```
RecordType = RECORD FieldListSequence END.
FieldListSequence = FieldList { ";" FieldList }.
FieldList = [ IdentList ":" Type ].
IdentList = IdentDef { "," IdentDef }.
```

If a record type is exported, all field identifiers in it are made visible outside the declaring module. A more selective export, like that of Oberon will be implemented in the next version of lo. Examples of record types:

```
ARRAY[0..tmax] OF
  RECORD
     name:alfa;
     kind: BOOLEAN;
     level, val, adr: INTEGER
  END;
```

6.5 Pointer Types

Variables of a pointer type `P` assume as values pointers to variables of some type `T`. The pointer type `P` is said to be *bound* to `T`, and `T` is the *pointer base type* of `P`. `T` must be a record, an array, or a set type.

```
PointerType = ^Type.
```

If `p` is a variable of type `P = ^T`, then a call of the predefined procedure `NEW(p)` has the following effect: a variable of type `T` is allocated in free storage, and a pointer to it is assigned to `p`. This pointer `p` is of type `P`; the *referenced* variable `p^` is of type `T`. Failure of allocation results in `p` obtaining the value `NIL`. To any pointer variable may be assigned the value `NIL`, which points to no value at all. A call to the predefined procedure `FREE(p)` frees the space taken by the variable `p`; an error occurs if `p` is `NIL`.

6.6 Subprogram (Procedure/Function) Types

A subprogram type is a (named) type declaring a subprogram signature (heading).

```
SubprogramType = FunctionType | ProcedureType.
FunctionType = FUNCTION "(" [FormalParameters] ")" ":" FormalType.
ProcedureType = PROCEDURE [ "(" FormalParameters ")"].
FormalParameters = FPSection { ";" FPSection } ].
FPSection = [ VAR ] ident { "," ident } ":" FormalType.
FormalType =  QualIdent.
```

Variables of a procedure type `T` have a procedure as value. Anonymous subprogram types are not allowed in variable declarations. For example the following declaration generates an error message:

```
VAR p: PROCEDURE(VAR a: REAL; b:INTEGER); (* error *)
```

The above *non-orthogonal* restriction derives from the strict name-equivalence adopted in |O for assignment equivalence: no assignment compatible procedure could be defined otherwise (like in parameter list specification). A correct implementation of the above example is:

```
TYPE PP = PROCEDURE(VAR a: REAL; b:INTEGER);
VAR p: PP;
```

7 Variable Declarations

A variable declaration is used to introduce an identifier naming a variable and to associate it with a data type. A variable name must be unique within a given scope. Variables are allocated in computer memory and can hold values assigned to them.

```
VariableDeclaration = IdentList ":" Type.
```

Variables whose identifiers appear in the same variable declaration IdentList (e.g., `k`, `k1`, `n` in the below snippet of code) or that appear in different IdentList using the same type identifier (e.g., `n`, and `c1` or `id`, and `a` in the below snippet of code) are (by pair) of the same type; they are considered of different type otherwise (e.g., `aaa`, `bbb` in the below snippet of code). Examples of variable declarations (refer to the type declarations in section 6):

```
z, w: REAL;
root: wordref;
suspended: BOOLEAN;
k, k1, n: INTEGER;
id: alfa; ch, cho: CHAR;
f, g: TEXT;
a: alfa; c1: INTEGER;
mnemonic: ARRAY[0..7] OF ARRAY[1..5] OF CHAR;
declbegsys, statbegsys, facbegsys, mysys: symset;
table:  ARRAY[0..tmax] OF
          RECORD
            name:alfa;
            kind: INTEGER;
            level,val,adr: INTEGER;
          END;
aaa: ARRAY [1..100] OF REAL;
bbb: ARRAY [1..100] OF REAL;
```

8 Statements

Statements specify the actions to be performed by a program. There are *simple* and *compound* statements. Simple statements are not composed of any parts that are themselves statements. They are the assignment, the procedure call, the return, cycle and

exit statements. Compound statements are composed of parts that are themselves statements. They are bracketed by keywords giving a clear indication of the kind of sequencing, conditional, selective, and repetitive execution to be performed. A statement may also be empty, in which case it denotes no action. The empty statement is included in order to relax punctuation rules in statement sequences.

```
Statement = [ Assignment | ProcedureCall | IfStatement |
   CaseStatement | WhileStatement | DoUntilStatement |
   ForStatement | DoLoopStatement | WithStatement |
   EXIT | CYCLE | RETURN [ Expression ] ].
```

8.1 Assignments

The assignment computes the value of an expression and stores the value in a variable. The assignment operator is written as "`:=`" and pronounced as "becomes."

```
Assignment = Designator ":=" Expression.
```

A *designator* is an expression denoting a storage location. The assignment statement must satisfy the following requirement: the type of the right-hand-side (RHS) expression must be *assignment compatible* to the type of the left-hand-side (LHS) designator, and its value must be a member of the left-hand designator type. The LHS designator type is *assignment compatible* with the RHS expression type if:

❶ They are the same type;

❷ the LHS type is `REAL` and the RHS type is `INTEGER`;

❸ they are both `SET` types with the same base type;

❹ the LHS type is `ARRAY[1..n] OF CHAR` and the RHS is a string value of length less or equal to *n*;

❺ the LHS type is a *pointer* and the RHS value is `NIL`.

Strings can be assigned to any variable whose type is an array of characters, provided the length of the string is equal (or less) than that of the array. Examples of assignments (see variable declarations in section 7):

```
table[c1].name:='MyOwnName '
suspended := ch = cho;
z := w + 1;
w := LN(1.44);
facbegsys := [1, 2, 14, 18. .22];
aaa[i] := w - z;
```

8.2 Subprograms Activation

Subprograms are activated by means of subprogram calls. A procedure is activated by a procedure-call, while a function is activated by specifying its designator in an expression. A subprogram call may contain a list of *actual parameters* that must match in

position, number and type to the corresponding formal parameters declared in the subprogram header of the procedure declaration (see section 9). Parameter-less functions are specified with an empty parameter list denoted by a pair of empty parenthesis `()`; parameter-less procedures are specified by their name standing alone.

Actual parameters are evaluated before the call and are substituted in place of their corresponding formal parameters.

There are two *modes* of parameter passing: by *variable* and by *value*. For a formal variable parameter of mode variable the corresponding actual parameter must be a designator denoting a variable whose type is assignment compatible to the corresponding formal parameter type. If it designates an element of a structured variable, the selector is evaluated when the formal/actual parameter substitution takes place, i.e., before the execution of the procedure. If the formal parameter has mode value, the corresponding actual parameter must be an expression which is evaluated prior to the procedure activation, and the computed value is assigned to the formal parameter which behaves, in the subprogram body, like a local variable (see also section 9.1).

```
ProcedureCall = Designator [ ActualParameters ].
```

Examples of procedure calls:

```
VarCall[i](x,y)
WRITELN('value=',j+3*i)
SIN(x)
```

8.3 Statement Sequences

Statement sequences specify a series of actions to be carried out, one after the other, in the order specified.

```
StatementSequence = Statement { ";" Statement }.
```

8.4 If Statements

```
IfStatement = IF Expression THEN StatementSequence
            { ELSIF expression THEN StatementSequence }
            [ ELSE StatementSequence ]
              FI.
```

If statements specify the conditional execution of one of a sequence of statements, depending on the value of one or more boolean expressions. The boolean expressions are evaluated in sequence of occurrence, until one evaluates to `TRUE`, where after its associated statement sequence is executed. If none of the boolean expressions is satisfied, the statement sequence following the symbol `ELSE` is executed, if there is one.
Example:

```
IF ch IN ['A'..'Z'] THEN BuildId
ELSIF ch IN ['0'..'9'] THEN BuildNum
ELSIF (ch >= 0X) AND (ch <= 020X) THEN ProcessControl
ELSE DoOtherWise
FI
```

8.5 Case Statements

A case statements is used to select and to execute one of a number of statement sequences based on the value of an expression. First, the case expression is evaluated, then the statement sequence which is preceded by a case label list containing the computed value is executed. The case expression and all the labels must be of the same integer or char type. The case labels are constants, with no value occurring more than once. If the value of the expression does not occur as a label of any case, the statement sequence following the symbol ELSE is executed if present, otherwise an error condition is raised.

```
CaseStatement = CASE Expression OF Case { "\" Case }
   [ ELSE StatementSequence ] FO.
Case = [ CaseLabelList ":" StatementSequence ].
CaseLabelList = CaseLabels { "," CaseLabels }.
CaseLabels = ConstExpression [ ".." ConstExpression ].
```

Example:

```
CASE ch OF
  'A' .. 'Z': ProcessLetter
\ '0' .. '9': ProcessNumber
  ELSE DoOther
FO
```

8.6 While Statements

A while statement specifies the repeated execution of a statement sequence under the control of a boolean expression. The expression evaluation and the statement execution are repeated as long as the boolean expression yields the value TRUE. If the initial evaluation of the boolean expression yields the value FALSE, the while statement is equivalent to a null statement.

```
WhileStatement = WHILE Expression DO StatementSequence OD.
```

Examples:

```
WHILE sym IN [plus,minus]
DO
  addop:=sym;
  getsym;
  term(fsys+[plus,minus]);
  IF addop=plus THEN gen(opr,0,2) ELSE gen(opr,0,3) FI
OD
```

8.7 Do-Until Statements

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

```
DoUntilStatement = DO StatementSequence UNTIL Expression.
```

Example:

```
id := a; i := 1; j := norw;
DO k := (i+j) DIV 2;
   IF id <= word[k] THEN j := k-1 FI;
   IF id >= word[k] THEN i := k+1 FI;
UNTIL i > j;
IF i-1 > j THEN sym := wsym[k] ELSE sym := ident FI
```

8.8 For Statement

A for statement specifies the repeated execution of a statement sequence while stepping a *control variable* upward or downward by one from a starting value to an ending value. The control variable must be local to the enclosing block.

```
ForStatement = FOR Ident ":=" Expression (TO|DOWNTO) Expression
DO StatementSequence OD.
```

The statement

```
FOR v := beg TO end DO statements OD
```

is equivalent to

```
v := beg;
WHILE v <= end DO statements; v := v+1 OD
```

The statement

```
FOR v := beg DOWNTO end DO statements OD
```

is equivalent to

```
v:=beg;
WHILE v >= temp DO statements; v := v-1 OD
```

Examples:

```
FOR i := 0 TO 79 DO k := k+a[i] OD
FOR i := 79 DOWNTO 1 DO k := k+a[i-1] OD
```

8.9 Do-Loop Statements

A do-loop statement specifies the endless repeated execution of a statement sequence. The loop can be terminated by the execution of any exit statement within that sequence (see section 8.10).

```
LoopStatement = DO StatementSequence OD.
```

Example:

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

8.10 Return, Exit and Cycle Statements

The return statement terminates the execution of a subprogram. If the subprogram is a function the return statement must be followed by an expression providing the result value computed by the function. The return statement indicates the termination of a procedure or function, with the expression specifying the result of a function. Its type must be identical to the result type specified in the procedure heading (see section 9).

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

An exit statement consists of the symbol `EXIT`. It specifies termination of the enclosing loop statement and continuation with the statement following that loop statement. A cycle statement consists of the symbol `CYCLE`. It specifies the termination of the current iteration of the most deeply enclosing loop statement, and the starting of a new one: all statements between the `CYCLE` statement and the end of the loop are skipped and the loop is continued from its entry point.

Exit and cycle statements are contextually, although not syntactically bound to the loop statement which contains them; a program can be terminated in every context by a call to the predefined parameter-less procedure `HALT`.

9 Subprograms

A subprogram may be either a *procedure* either a *function.* A procedure is used to perform an action, while a function is used to calculate a value. A subprogram is composed by a *heading (signature/prototype)*, and a *body.* When the declaration of a subprogram is at the top-level of a module it is called a top-level subprogram. The heading specifies the subprogram identifier, the *formal parameters*, and the result type (for functions). The body contains declarations and statements. The procedure identifier may be repeated at the end of the procedure declaration for increasing program readability.

Function subprograms are activated by a function designator as a constituent of an expression; they yield a result that acts as an operand in the expression. Procedures are activated by a procedure call. The function subprogram is distinguished in the declaration by indication of the type of its result following the parameter list. Its body

must contain a RETURN statement which defines the result of the function procedure and terminates the body execution.

In |O there are two kind of subprograms: *assignable* subprogram and *non-assignable* subprogram. Assignable subprograms are subprograms of named type, while non-assignable subprograms have an anonymous type — they cannot be assigned to variables because of type incompatibility. Example:

```
TYPE F = FUNCTION(x: REAL): REAL;
VAR vf: F;

FUNCTION f: F;
  (* body of f*)
END f;

FUNCTION g(x: REAL): REAL;
  (* body of g*)
END g;

vf := f; (* OK same type *)
vf := g; (* error different types*)
```

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

In addition to its formal parameters and locally declared entities, the entities declared in the environment of the procedure are also visible in the procedure (with the exception of those entities that have the same name as an object declared locally). The use of the procedure identifier in a call within its declaration implies recursive activation of the procedure.

```
SubprogramDeclaration = FullDeclaration | ForwardDeclaration |
                        ForwardCompletion.
FullDeclaration = SubprogramHeading ";" SubprogramBody [Ident].
SubprogramHeading = ProcedureHeading | FunctionHeading.
ProcedureHeading = NamedProcheading | AnonymousProcHeading.
NamedProcheading = PROCEDURE IdentDef: ProcedureType.
AnonymousProcHeading = PROCEDURE IdentDef FormalParameters.
FunctionHeading = NamedFuncHeading | AnonymousFuncHeading.
NamedFuncHeading = FUNCTION IdentDef: FunctionType.
AnonymousFuncHeading = FUNCTION IdentDef FormalParameters ":" Resultype.
SubprogBody = DeclarationSequence CompoundStatement.
DeclarationSequence = { CONST { ConstantDeclaration ";" } |
                        TYPE { TypeDeclaration ";" }      |
                        VAR { VariableDeclaration ";"
                      } { SubprogramDeclaration ";" }.
CompoundStatement = "BEGIN" StatementSequence "END".
ForwardDeclaration  =  (ForwardProcHeading|ForwardFuncHeading)";".
ForwardProcHeading = PROCEDURE "^" IdentDef FormalParameters.
ForwarFuncHeading = FUNCTION "^" IdentDef FormalParameters
```

```
                              ":" Resultype.
ForwardCompletion    =    (PROCEDURE|FUNCTION) IdentDef ";"
                          SubprogramEnv ";"SubprogramBody [Ident].
```

A *forward declaration* supports forward references to a subprogram that appears later in the text in form of *forward completion.* A forward subprogram definition is broken in two parts: a forward heading which specify the kind of a subprogram, the name preceded by the forward marker "^", the formal parameters if any, and a result type for functions. Later and in the same scope, a *forward completion* clause has to be provided for completing the subprogram declaration by including an environment and a body. The forward completion starts with short heading recalling the kind (function or procedure) and the name of the forward declaration to be completed (formal parameters and result type must be omitted).

9.1 Formal Parameters

Formal parameters are identifiers which are *bound* to the actual parameters specified in each subprogram call. The binding is performed when the procedure is called. There are two modes of passing parameters, namely by *value* and by *variable*. The mode is indicated in the formal parameter list. Value parameters stand for local variables to which the result of the evaluation of the corresponding actual parameter is assigned as initial value. Variable parameters correspond to actual parameters that must be variables, and they stand for these variables. Variable parameters are indicated by the symbol `VAR`, value parameters by the absence of the symbol `VAR`. A function procedure without parameters may have an empty parameter list expressed by a pair of matching parenthesis following its name. It must be called by a function designator whose actual parameter list is written in the same way.

Formal parameters belong to the procedure environment (they are local to the procedure), i.e., their scope is the program text which constitutes the procedure declaration.

```
FormalParameters = [ "(" FPSection { ";" FPSection } ")" ]
   [ ":" QualIdent ].
FPSection = [ "VAR" ] ident { "," ident } ":" FormalType.
FormalType = QualIdent.
```

The type of each formal parameter is specified in the parameter list. For variable parameters, it must be identical to the corresponding actual parameter's type, except in the case of a record, where it must be a base type of the corresponding actual parameter's type. For value parameters, the rule of assignment holds (see section 8.1). If a formal parameter specifies a procedure type, then the corresponding actual parameter must be either a procedure declared at the same level (level 0) or a variable (or parameter) of that procedure type. It cannot be a predefined procedure. The result type of a procedure can be either a record either an array. Examples of procedure declarations:

```
TYPE alfa = ARRAY[1..c1] OF CHAR;
  wordref = ^word;
  itemref = ^item;
  word = RECORD
```

```
    key: alfa;
    first, last: itemref;
    left, right: wordref
  END;
PROCEDURE printtree(w: wordref);
  PROCEDURE printword(w: word);
  VAR l: INTEGER; x:itemref;
  BEGIN
    WRITE(' ', w.key);
    WRITE(g, ' ', w.key);
    x := w.first; l := 0;
    DO
      IF l=c2 THEN
        WRITELN; WRITELN(g); l := 0;
        WRITE(' ':c1+1); WRITE(g,' ':c1+1)
      FI;
      l := l+1; WRITE(x.ln0:c3);
      WRITE(g,x.ln0:c3); x := x.next;
    UNTIL x = NIL;
    WRITELN; WRITELN(g);
  END; (*printword*)
BEGIN (*printree*)
  IF w <> NIL THEN
    printtree(w.left);printword(w^);printtree(w.right)
  FI
END; (*printtree*)
```

9.2 Predefined Subprograms

The following table lists the predefined subprograms. Most subprograms are *polymorphic*, i.e., they apply to several types of operands. `v` stands for a variable, `x` and `n` for expressions, and `T` for a type.
Numeric, utility and I/O functions:

Name	*Argument type*	*Result type*	*Function*
`ABS(x)`	numeric type	type of `x`	absolute value
`ODD(x)`	integer type	`BOOLEAN`	`x MOD 2 = 1`
`SQR(x)`	numeric type	numeric type	x^2
`SIN(x)`	`REAL`	`REAL`	sin(x)
`COS(x)`	`REAL`	`REAL`	cos(x)
`EXP(x)`	`REAL`	`REAL`	exp(e)
`LN(x)`	`REAL`	`REAL`	$\log_e(x)$
`SQRT(x)`	`REAL`	`REAL`	$\sqrt{x}$
`ARCTAN(x)`	`REAL`	`REAL`	arctan(x)

Type conversion functions:

Name	*Argument type*	*Result type*	*Function*
`ORD(x)`	ordinal type	`INTEGER`	ordinal number of `x`
`CHR(x)`	integer type	`CHAR`	character with ordinal number `x`
`ROUND(x)`	`REAL INTEGER`	`INTEGER`	closest integer
`TRUNC(x)`	`REAL INTEGER`	`INTEGER`	discard fractional part

Procedures for memory, and reflective shift-up/down management:

Name	*Argument type*	*Function*
`NEW(v[,s])`	pointer type, size	allocate `v^of size s`
`FREE(v)`	pointer type	deallocate `v^`
`HALT`		terminate program execution
`IDLE[(v)]`	`INTEGER constant`	suspend component execution
`INT[(v)]`	`INTEGER constant`	resume component execution

10 Input/Output

lo implements a built-in I/O facility, the sequential ASCII files called the *text-streams* (text for short) of Pascal. Texts are a predefined type that can be opened in two modes *read* and *write* by means of the following primitives:

`RESET(x,fnam,fext)`	*opens a text-stream for reading (mode read)*
`REWRITE(x,fnam,fext)`	*opens a text-stream for writing (mode write if the stream exists it is replaced with an empty stream.*

Syntax:

```
RESET"(" [StreamDesignator ","] {FileName ["," FileExtension]]")".
REWRITE"(" [StreamDesignator ","] {FileName ["," FileExtension]]")".
```

Examples:

```
RESET(MyFile,'DataIn','dat')
```

The following predefined procedures:

`READ([x,]vl)`	*reads data from the input stream*
`WRITE([x,]el)`	*writes data to the output stream*
`EOF[(x)]`	*senses the end of file, it holds `TRUE` if the current file position is at the end of the input stream. `FALSE` otherwise.*
`CLOSE(x)`	*is a procedure for closing the connection with a stream.*

Syntax:

```
READ"(" [StreamDesignator ","] VarDesignator{"," VarDesignator}")".
WRITE"(" [StreamDesignator ","] OutputExpression{"," OutputExpression}")".
OutputExpression = Expression [":" Expression [":" Expression]].
EOF["("StreamDesignator")"]
CLOSE "("StreamDesignator ")"
```

Text-streams are structured in *lines* and IO provides three primitives for managing lines of text:

`READLN([x,]vl)`	*reads data from the input stream up to the end of line*
`WRITELN([x,]el)`	*writes data to the output stream and opens a new line*
`EOLN[(x)]`	*senses the end of line, it holds `TRUE` if the current file position is at the end of the input line, `FALSE` otherwise.*

`READLN(fil,v1,...,vn)` is equivalent to: `READ(fil,v1,...,vn); READLN(fil).` While `READLN(fil)` will cause a skip to the beginning of the next line.
`WRITELN(fil,e1,...,en)` is equivalent to: `WRITE(fil,e1,...,en); WRITELN(fil).`

11 Modules

A program in IO is a collection of few (in general no more than 3) reflective and executable components. Each component is composed by a collection of separately compiled modules. In turn, a module is a composed by set of declarations and a sequence of statements.

```
Module = MODULE Ident["("Pragma{","Pragma}")"] ";"
[UseList]
[GlobList]
DeclarationSequence
BEGIN [StatementSequence ]
END [Ident]".".
UseList = QualifiedUseList | UnqualifiedUseList.
UnqualifiedUseList = OF Ident USE Use { "," Use } ";".
QualifiedUseList = USE Use { "," Use } ";".
Use = Ident [AS Ident].
GlobList = GLOBAL Ident { "," Ident} ";".
Pragma = Io_Sys_ | TABLES ...
```

Identifiers that are to be visible in client modules, i.e., outside the declaring module, must be listed in a GlobList declaration. The UseList specifies the identifiers declared in other modules and used within the module. The global identifiers listed in a unqualified UseLists (they are declared in module ModulId) may be directly referred in the code. In a qualified UseList all the identifiers of the list are interpreted as module names: all the global identifiers declared in such modules must be accessed in qualified notation (i.e. prefixed by the exporting module name). If an identifier `x` is exported by a module `M`, and `M` is listed in a module's import list, then `x` is referred to as `M.x`. If

the form `M AS M1` is used in the use list, that object `x` declared within `M` is referenced as `M1.x`. Moreover, if an identifier `x` is listed in a unqualified UseList in the form `x AS y` then the object `x` will be referred as `y`. Identifiers that are to be visible in client modules, i.e., outside the declaring module, must be listed in a GlobList declaration. The statement sequence following the symbol `BEGIN` is executed when the component is loaded. Example of module (see Appendix A where `Io_Env` is shown):

```
MODULE Io_Env_Tst;
OF Io_Env USE alng_, tmax1_, cmax_, wksize_,
   alph_, sgtyp_, ordtyp_, sgt_, code_, tt_, t00_, csiz_, t_, b_, hb_, ws_,
   maxf_, curf_, dfnam_, dfext_, ir_, pc_, ps_, lmax_, lncnt_,
   ocnt_, blkcnt_, chrcnt_, fld_, dspy_;
VAR j: INTEGER; outff: TEXT; filnam: ARRAY[1..6] OF CHAR;

PROCEDURE writetables(i1: INTEGER);
VAR i, i2: INTEGER;
BEGIN
  (* body of the procedure *)
END (*writetables*);

PROCEDURE writecode;
VAR i: INTEGER;
BEGIN
  WRITELN(outff, 'code');
  FOR i:=0 TO csiz_ DO
    WRITELN(outff, code_[i].f, code_[i].x, code_[i].y)
  OD
END (*writecode*);

BEGIN (*main*)
  WRITE('progname>');
  READ(filnam);
  REWRITE(outff, filnam, 'cic');
  writetables(t00_);
  writecode;
  FOR j := 1 TO 7 DO WRITELN(outff, fld_[j]) OD
END.
```

11.1 Module Inheritance and Interfaces

Modules can also inherit from other modules. Inheritance is obtained by exporting imported entities in a `GLOBAL` clause. Such entities *gain* a *definition status* that equates them to entities defined into the module itself, thus they can be exported. However, in each executable component a module is instantiated only once; it follows that the direct use of an object imported from the defining module is equivalent to its indirect use through an intermediate inheriting module because all references related to the same module are unified by connecting them with the unique instance of the module included into each importing component. lO compiler maintains in its symbol tables

information for identifying all the instances of a module even if they have been renamed (clause AS). Example:

```
OF M USE a, b, c;
GLOBAL a, b;
```

In this example a and b are *inherited*, i.e. they are made accessible to the importing modules.

12 Expressions

Expressions are constructs for computing a value: they specify operators and data to be operated upon. The simplest expression is composed by:

- The name of a variable, constant or subprogram and is called a *designator.*
- A literal value such as a number, a character or a string.

Parentheses may be used to express specific associations of operators and operands.

12.1 Operands

A designator may possibly be qualified by module identifiers (see sections 9 and 8.2), and it may be followed by selectors, if the designated object is an element of a structure.

The notation A[E] designates the element of the array A denoted by the value of the expression E. The type of E must be an ordinal type. The form A[E1, E2, ..., En] is a shorthand of A[E1][E2]...[En]. If r is a record, then r.f denotes the field f of r. If p^ is a record, then p.f denotes the field f of p^, i.e., the dot implies dereferencing and p.f stands for p^.f; if p^ is an array then p[E] denotes the element of p^ indexed by E.

```
Designator = QualIdent { "." Ident | "[" ExpressionList "]" |
   "(" QualIdent ")" | "^" }.
ExpressionList = Expression { "," Expression }.
```

If the designated object is a variable, then the designator refers to the variable's current value. If the object is a function, a designator without parameter list refers to that function. If it is followed by a (possibly empty) parameter list, the designator implies an activation of the function and stands for the value resulting from its execution. The (types of the) actual parameters must correspond to the formal parameters as specified in the procedure's declaration (see section 8.1). Examples of designators (they refer to the examples in section 7) are:

```
k                    (INTEGER)
a[k]                 (CHAR)
table[n].nam         (alfa)
table[n].val         (INTEGER)
t.key                (INTEGER)
w                    (REAL)
mnemonic[k1]         (ARRAY[1..5] OF CHAR)
```

12.2 Syntax of Expressions and Operators

The syntax of expressions is built over four classes of operators with four different precedences (binding strengths). The operator NOT (~) has the highest precedence, followed by the multiplicative, the additive and finally the relational operators. Operators of the same precedence associate from left to right. For example, h-l-m stands for (h-l)-m.

```
Expression = SimpleExpression [ Relation SimpleExpression ].
Relation = "=" | "<>" | "<" | "<=" | ">" | ">=" | IN.
SimpleExpression = [ "+" | "-" ] Term { AddOperator Term }.
AddOperator = "+" | "-" | OR.
Term = Factor { MulOperator Factor }.
MulOperator = "*" | "/" | DIV | MOD | AND
Factor = Number | CharConstant | String | NIL | Set |
   Designator [ ActualParameters ] | "(" Expression ")" | NOT Factor.
Set = "[" [ Element { "," Element } ] "]".
Element = OrdinalConstant [ ".." OrdinalConstant ].
OrdinalConstant= Char | Integer.
ActualParameters = ["(" [ ExpressionList ] ")"].
```

12.2.1 Boolean Expressions

Boolean values are computed by means of relations or by combining boolean values with the following logical operators

symbols	*result*
OR or |	logical disjunction
AND or &	logical conjunction
NOT or ~	negation

These operators apply to BOOLEAN operands and yield a BOOLEAN result.

p OR q	stands for "if p then TRUE, else q"
p AND q	stands for "if p then q, else FALSE"
NOT p	stands for "not p"

12.2.2 Set Expressions

A set is an unordered collection of elements all of the same type. A set may be empty (denoted []). The elements of a set must be values of an ordinal type (i.e. a subset of type INTEGER or CHAR). Sets may be operated with the following operators:

symbol	*result*
+	union
-	difference
*	intersection

Examples

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

12.2.3 Arithmetic Expressions

Arithmetic expressions are obtained by combining arithmetic values and variables with the following operators

symbol	*result*
`+`	sum
`-`	difference
`*`	product
`/`	quotient
`DIV`	integer quotient
`MOD`	modulus
`OR` or `|`	integer bitwise AND
`AND` or `&`	integer bitwise OR
`NOT` or `~`	integer bitwise NOT

Every operation has a type, which determines the value it can take and the operations that can be performed on that value. The operators `+`, `-`, `*`, and `/` apply to operands of numeric types. The type of the result is that operand's type which includes the other operand's type. When used as operators with a single operand, `-` denotes sign inversion and `+` denotes the identity operation. The logical operators `AND` (`&`), `OR` (`|`) and `NOT` (`~`) apply to integer operands and produce bitwise *and, or* and *not* integer results. The division (`/`) operator applies to numeric types and produces a result of real type. The operators `DIV` and `MOD` apply to integer operands only. They are related by the following formulas defined for any dividend `x` and positive divisors `y`:

$$x = (x \texttt{ DIV } y) * y + (x \texttt{ MOD } y), \textit{ and } 0 \leq (x \texttt{ MOD } y) < y$$

12.2.4 Relations

Relations are composed by two expressions of the same scalar type (for the operator `IN` if the first has type `T` then the second must have type `SET OF T`) compared to form a truth value by one of the following relational operators:

symbol	*result*
`=`	equal
`<> or #`	unequal
`<`	less
`<=`	less or equal
`>`	greater
`>=`	greater or equal
`IN`	set membership

Relations are special cases of Boolean expressions. The ordering relations `<`, `<=`, `>`, and `>=` apply to the numeric types, `CHAR`, and alpha-strings. Moreover, `=` and `<>` also apply to the type `BOOLEAN`, `SET`, `POINTER`, and subprogram types. `x IN s` stands for "`x` is an element of `s`." `x` must be of an ordinal type `T` (`INTEGER` or `CHAR`), and `s` of type `SET OF T`. Examples of expressions (refer to examples in 7):

```
n DIV 3                        (INTEGER)
~suspended|(ch=cho)            (BOOLEAN)
(i+j) * (i-j)                  (INTEGER)
mysys - [8, 9, 13]             (SET)
bbb[k1] + aaa[n]               (REAL)
(0<=k)AND(k<100)               (BOOLEAN)
table[4].name='AlphaString'    (BOOLEAN)
k IN facbegsys                 (BOOLEAN)
```

References

[1] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[2] Brian C. Smith. Reflection and Semantics in a Procedural Language. Technical Report 272, MIT Laboratory of Computer Science, 1982.

A Reflective Programming

|O programs are intended to be executed in a variety of applicative environments ranging from embedded real-time systems to environments for distributed applications. The difference concerns the way programs are designed and developed by focusing the attention on the operative environment and on reflective computing. Computational reflection is the activity performed by a computational agent when performing computations about its own computation. |O has a Reflective Run-Time Execution environment (RRTE) which endows reflection into its basic semantics.

The design of a program starts with the localization of the *causality domains.* A computational system S_1 (with computational domain D_1) is causally connected with a computational system S_0 (with domain D_0) if the internal data structures of D_1 represent computations in S_0 and any change on D_1 is *reflected* in a corresponding effect on system S_0. S_1[4] is called the meta-level. The design is iterated for pointing out an eventual meta-meta-level S_2 and so on. In this way, each application program is decomposed into *reflective components* $S_i, i = 0, \ldots, n$. If $n = 0$ then the program is not reflective and organized into a unique component S_0.

Once the layers have been determined, their mutual *intra-layers* interface has to be designed by defining the operations of each meta-level and the corresponding activation mechanisms (or shift-up methods) to be used for their activation by the interpreter or by the machine trap/ interrupt system (|O supports both external interrupts or software supervisor calls inserted into the code); more sophisticate interface may be built over it. As a consequence of the above schema, the translation of an |O program into an executable image is different from a traditional compilation process.

[4] We reverse the traditional reflective tower in |O: level 0 is the non-reflective application layer, while layers 1,2,.. are the *meta-level*, *meta-meta-level* etc.

B The Module Io_Sys

The module Io_Sys contains the definition of the basic type Io_Word describing each word of the working storage as implemented by the RTS interpreter (Ioi) of |O. The type Io_Word has the following definition, exported by the system level module Io_Sys. Note that |O does not support variant parts of a record, only the system type Io_Word has this property and it is implemented in a special way.

```
TYPE
  Io_Word = RECORD
    CASE (ints, reals, bools, chars) OF
      ints: (i: INTEGER)
    \ reals: (r: REAL)
    \ bools: (b: BOOLEAN)
    \ chars: (c: CHAR)
    FO
   END;
```

C The Module Io_Env

The module Io_Env exports the definition of all basic low-level reflective RRTE data structures implemented by the Ioi interpreter (including Io_Word inherited from Io_Sys). The structure of Io_Env is the following:

```
MODULE Io_Env(Io_Sys_);
(* This program maps all the low-level data structures of the Io *)
(* interpreter implementing the reflective tower to the importing *)
(* reflective components. Names are filled of underline chars for *)
(* minimizing name clashing. Io_Env inherits Io_Word from Io_Sys *)
OF Io_Sys USE Io_Word;
GLOBAL Io_Word, alng_, tmax1_, cmax_, wksize_, alph_, sgtelem_, ordtyp_,
      SgtTyp_, CodTyp_, TwrElem_, TwrTyp_, TWR_;
(* Compiler/Interpreter dependencies: alng_, tmax1_, cmax_, wksize, *)
(* maxf, lmax_ *)
CONST alng_=12; tmax1_=2001; cmax_=30000; wksize_=60000;
     maxf_=6; lmax_=15; Layers_=2; (* number of reflective layers-1*)
TYPE alph_=ARRAY[1..alng_] OF CHAR;
    sgtelem_=RECORD
         nam_: alph_; mdl_, nxt_, knd_, typ_, xtp_: INTEGER;
         ref_, nrm_, lev_, adr_, xrf_, rsz_, sts_, rel_: INTEGER;
     END;
     ordtyp_=RECORD
              f, x, y: INTEGER
             END;
     SgtTyp_ = ARRAY[0..tmax1_] OF sgtelem_;
     CodTyp_ = ARRAY[0..cmax_] OF ordtyp_;
(*Mirrored Comp. Domain *)
     TwrElem_ = RECORD (* one element of the reflective tower *)
                  sgt_: SgtTyp_; (* symbol table = semantic Env.*)
```

```
                code_: CodTyp_; (* executable instructions*)
                tt_, t00_, csiz_, t_, b_, hb_: INTEGER;
                curf_1:INTEGER;
                ir_:ordtyp_;     (*instruction buffer*)
                pc_, ps_ :INTEGER; (* pgm counter & status regs*)
                lncnt_, ocnt_, blkcnt_, chrcnt_ :INTEGER;
                dspy_:ARRAY[0..lmax_] OF INTEGER; (*display *)
                dfnam_, dfext_:ARRAY[0..maxf_] OF alph_; (* Iostreams*)
                fld_:ARRAY[1..8] OF INTEGER;(* Iostreams*)
                ws_: ARRAY[1..wksize_] OF Io_Word;(*workstore=semantic store*)
               END;
    TwrTyp_ = ARRAY[0..Layers_] OF TwrElem_; (* the reflective tower type *)

(* the interpreter sets dspy_[0]:=sgtbase[rlev+1] *)

VAR     TWR_: TwrTyp_; (* the virtual reflective tower *)
BEGIN
END.
```

A detailed description of the data structures of `Io_Sys` and `Io_Env` can be found in the `IoVM` virtual machine report.

D Syntax

The following EBNF specification collects all syntactic rules presented in the report[5].

```
Module = MODULE Ident["("Pragma{","Pragma}")"] ";"
         [UseList] [GlobList] DeclSeq
         BEGIN [StatementSequence] END [Ident]".".
UseList = QualUseList | UnqualUseList.
UnqualUseList = OF Ident USE Use { "," Use } ";".
QualUseList = USE Use { "," Use } ";".
Use = Ident [AS Ident].
GlobList = GLOBAL Ident { "," Ident} ";".
Pragma = Io_Sys_ | TABLES ...
DeclSeq = {CONST {ConstDecl";"}| TYPE {TypeDecl";"}| VAR {VarDecl";"}|
          {SubProgDecl";"}.
ConstDecl = IdentDef "=" ConstExpr.
TypeDecl = IdentDef "=" Type.
Type =  ArrayType | RecordType | PointerType | SetType | SubprogType.
ArrayType = ARRAY "[" Index { "," Index }"]" OF Type.
Index = OrdinalConstant ".." OrdinalConstant.
SetType = "SET OF" OrdinalType.
OrdinalType = INTEGER | CHAR.
RecordType = RECORD
                FieldListSequence
             END.
FieldListSequence = FieldList { ";" FieldList }.
```

[5] Same syntactic names could have been abbreviated.

```
FieldList = [ IdentList ":" Type ].
IdentList = IdentDef { "," IdentDef }.
PointerType = ^ Type.
SubprogType = FunctionType | ProcedureType.
FunctionType = FUNCTION "(" [FormalParameters] ")" ":" FormalType.
ProcedureType = PROCEDURE [ "(" FormalParameters ")"].
FormalParameters = FPSection { ";" FPSection } ].
FPSection = [ VAR ] ident { "," ident } ":" FormalType.
FormalType =  QualIdent.
VarDecl = IdentList ":" Type.
SubProgDecl = FullDeclaration | ForwardDeclaration |
                        ForwardCompletion.
FullDeclaration = SubprogramHeading ";" SubprogramBody [Ident].
SubprogramHeading = ProcedureHeading | FunctionHeading.
ProcedureHeading = NamedProcheading | AnonymousProcHeading.
NamedProcheading = PROCEDURE IdentDef: ProcedureType.
AnonymousProcHeading = PROCEDURE IdentDef FormalParameters.
FunctionHeading = NamedFuncHeading | AnonymousFuncHeading.
NamedFuncHeading = FUNCTION IdentDef: FunctionType.
AnonymousFuncHeading = FUNCTION IdentDef FormalParameters ":" Resultype.
SubprogBody = DeclSeq CompoundStat.
CompoundStat = "BEGIN" StatementSequence "END".
StatementSequence = Statement { ";" Statement }.
Statement = [ Assignment | ProcedureCall | IfStatement |
              CaseStatement | WhileStatement | DoUntilStatement |
              ForStatement | DoLoopStatement | WithStatement |
              EXIT | CYCLE | RETURN [ Expression ] ].
Assignment = Designator ":=" Expression.
ProcedureCall = Designator [ ActualParameters ].
IfStatement = IF Expression THEN StatementSequence
            { ELSIF expression THEN StatementSequence }
            [ ELSE StatementSequence ] FI.
CaseStatement = CASE Expression OF Case { "\" Case }
              [ ELSE StatementSequence ] FO.
Case = [ CaseLabelList ":" StatementSequence ].
CaseLabelList = CaseLabels { "," CaseLabels }.
CaseLabels = ConstExpression [ ".." ConstExpression ].
WhileStatement = WHILE Expression DO StatementSequence OD.
DoUntilStatement = DO StatementSequence UNTIL Expression.
ForStatement = FOR Ident ":=" Expression (TO|DOWNTO) Expression
               DO StatementSequence OD.
LoopStatement = DO StatementSequence OD.
Designator = QualIdent { "." Ident | "[" ExpressionList "]" |
             "(" QualIdent ")" | "^" }.
QualIdent = [ Ident "." ] Ident.
ExpressionList = Expression { "," Expression }.
Expression = SimpleExpression [ Relation SimpleExpression ].
Relation = "=" | "<>" | "<" | "<=" | ">" | ">=" | IN.
SimpleExpression = [ "+" | "-" ] Term { AddOperator Term }.
AddOperator = "+" | "-" | OR.
```

```
Term = Factor { MulOperator Factor }.
MulOperator = "*" | "/" | DIV | MOD | AND
Factor = Number | CharConstant | String | NIL | Set |
Designator [ ActualParameters ] | "(" Expression ")" | NOT Factor.
Set = "[" [ Element { "," Element } ] "]".
Element = OrdinalConstant [ ".." OrdinalConstant].
OrdinalConstant= Char | Integer.
ActualParameters = ["(" [ ExpressionList ] ")"].
Ident = IdChar { IdChar | Digit }.
IdChar = Letter | "_".
Number = Integer | Real.
Integer = Digit { Digit } | Digit { HexDigit } "H".
Real = Digit { Digit } "." { Digit } [ ScaleFactor ].
ScaleFactor = "E" [ "+" | "-" ] Digit { Digit }.
HexDigit = Digit | "A" | "B" | "C" | "D" | "E" | "F".
Digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
CharConstant = "'" Char "'" | Digit { HexDigit } "X".
String = ' { CharN | "''" } '.
Char = CharN | "'".
```