transformations de programmes

3e colloque international sur la programmation

005.106
I61pr

phase recherche

DUNOD
informatique

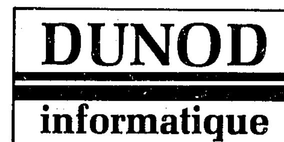
transformations de programmes

program transformations

Actes du 3[e] colloque international sur la programmation
direction B. Robinet
Paris
28 - 30 mars 1978

Proceedings
of the 3rd international symposium on Programming
edited by B. Robinet
Paris
March, 28 - 30 1978

phase recherche



INTRODUCTION

In order to do any work on a computer nowadays the user has to learn a second language: a control language of the operating system used. By means of requests in this language, the user commands the execution of object programs within a environment of data files determined by himself, and at the same time requests resources (processor time, main store, file space, output limits) for his programs.

As has been commented [Barron and Jackson, 1972] control languages are themselves programming languages since they possess control and data structures akin to those of a programming language. However, whereas the state of the art of programming languages has advanced steadily, the same cannot always be said of control languages. The most widely used such language, IBM's Job Control Language (JCL) [IBM], which has remained virtually unaltered for over a decade, has been described as a macro language [Barron and Jackson, 1972]. A newer generation of control languages includes B. Landy's Phoenix [Cambridge, 1975] at the University of Cambridge (IBM 370/165 with modified OS/MVT), ICL's George languages, which have been described as forms of autocode [Barron and Jackson, 1972], and the Algol-like Work Flow Language (WFL) for the larger Burroughs machines.

Now the level of sophistication of a control language has a great deal to do with its ease of use by the average user, since he will generally construct his control program out of procedure calls (to edit, to compile, etc) and what is important is the ability to construct powerful procedures. This is impossible in JCL owing to the absence of looping, and the exceedingly primitive conditional commands available. On the other hand the Phoenix language offers a rich structure, and correspondingly powerful procedures. This structure includes the use of global and local variables of integer, Boolean and string types, conditional commands, and general branching, subroutine call and error handling facilities, in addition to the procedure call.

Unfortunately Phoenix is not a very systematic language particularly with regard to the scope of variable names. In any given procedure the variables in scope are those local to the procedure and any globals without local homonyms - there is no concept of nested scopes. Another problem is that there there is no checking of procedure parameters, which can lead to unexpected errors.

A CONTROL LANGUAGE FOR PROGRAMMERS

Michael Anthony Stanton (1)

ABSTRACT

Currently available control languages are briefly discussed, and a case is made for a well structured and powerful new language. There follows a description of the ARARA control language which seeks to include characteristics of a modern procedure oriented programming language. The dynamic nature of use of a control language has led to the introduction of a number of features, notably the inclusion of program text from an arbitrary file and the dynamic alteration of the execution time context by dynamic variable declarations and by the entry to and exit from "environments". Examples are given to illustrate the ideas.

---

(1) Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Gávea, Rio de Janeiro, Brasil.

The ARARA language described in this paper attempts to solve these and other problems which arise in the specification of a control language. A primary aim of this language is to follow closely the syntax of modern programming languages and thereby present the more advanced user with a familiar instrument. As the average user will mainly execute procedure calls, these must have a simple syntax. The description follows closely the Common Language for Utilities (CLU) proposed by P. Hazel [Hazel, 1976] but with certain important differences.

DESCRIPTION OF THE LANGUAGE

The overall view of the language is of a procedure oriented language in which all variables and procedures used have to be declared before use. The scope of a name is restricted to levels no higher than the level of nesting of its declaration, that is to say, Algol-like scope rules are used, except that declarations need not precede all other statements. This feature, like the others we shall describe, is necessary if the language is to be useful in an interactive environment.

The language supports the following data types:

integer, Boolean, proc, string, function, environment.

The type of a variable is dynamic and is checked dynamically.

The names of variables and procedures are chosen from a certain name space, from which certain elements are reserved for system words. Names (other than system words) may not be used without having been declared previously. Names may be declared in one of the following ways:

- variable declaration
- procedure or function definiton
- procedure call with arguments
- procedure call with local variable creation

- entering an environment

The scope of a name ends on leaving the nesting level of the procedure or environment in which it was declared. Scopes of homonyms are nested. Thus, only the most recently declared instance of the name is accessible.

The procedure call is unconventional in that the arguments may be either positional or identified by keyword. In addition, default values may be specified for omitted arguments in the procedure definition. Finally, the procedure call may declare and initialize variables local to the procedure. This facility may be used to override the value of a non-local variable for a given procedure call.

Consider the following example, in which a value is assigned to the variable ACCOUNT, and three calls are made to the procedure SUBMIT:

```
ACCOUNT := "0003"
SUBMIT, JOB1
SUBMIT, JOB2, ACCOUNT = "0025"
SUBMIT, JOB3
```

The second job would be submitted with account number 0025 and the others with account number 0003.

The language also permits the definition and use of functions which are similar to procedures, except in that they return a value. Thus they be used in value expressions.

There are occasions when a set of procedures may wish to comunicate amongst themselves. If the procedures are to be called sequentially then the common data area cannot be local to any of them. We thus introduce the concept of the environment, which maintains in its local variables the data common to its local procedures, very like the workspace concept of APL [Gilman and Rose, 1974] and somewhat akin to the monitor concept of Hoare [Hoare, 1974]. The use is different however, and this is by means of the environment entry, a statement similiar to the procedure call, in that it is nested within the containing procedure level. The effect of the environment entry is to alter the local context to include those variables and procedures declared within the environment. The old context may be restored by exiting from the environment.

Procedures, functions and environments have all to be declared before their use, by means of the appropriate declarations. To facilitate the manipulation of these and other sections of a control program text, there is provided a means to include program text from a data file in the command text or current procedure. A set of several environment declarations contained in a data file may be declared jointly in a library declaration.

The most fundamental command in a control language is the program execution command, which causes execution of an object program, previously compiled from some source language. Most operating systems already define a precise interface for calling such a program, which consists of the specification of data files used by the program and optional string data. In addition it is often necessary to specify the resources of processor time, main store or file space necessary for program execution. Finally the program may return a value, in the form of a return code under program control, or an abend code in the event of program failure. The program execution command provides a means of specifying all these arguments, and the value returned by the program is manipulated by the error handling procedures.

We should note here that the format of names of data files is operating system dependent. However there are some cases of special interest which can be distinguished. These include:

- the "immediate document" or "instream dataset", which consists of subsequent lines of the current command or procedure text up to a given delimiter , unprocessed in any way.

- as above, but with evaluation as text strings of integer or string variables whose names are enclosed in square brackets.

- the dummy file

- the standard output file

- the interactive terminal

For these we use the special filenames *IMMEDIATE(terminator), *SUBIMMEDIATE(terminator), *DUMMY, *OUTPUT and *TERM.

The nature of a control language requires that the user have adequate information about error conditions, to enable him to perform error handling under program control. This is achieved by the specification of <u>error procedures</u> to be called in the event of the occurrence of an error condition. Error conditions are grouped into several types, each of which is associated with a unique name. Thus, we may have

BADSYNT	syntax error
ABEND	command terminated abnormally
RC	command returned non-zero return code
BREAK	command interrupted by "break" at terminal

An error procedure is an ordinary procedure which is called automatically on occurences of the relevant type of error condition, and may be associated with a command, a procedure or an environment. In the event of an error condition, control passes to the nearest specified error procedure, which may either return control to the statement following the one causing the error, or else pass control up to the next level in which an appropiate error procedure is specified. In the event of control being passed up to the standard environment, the standard error function is called. In all cases the error procedure is entered as if it had been called by the statement

```
proc-name , ERRTYPE=error-type , ERRCODE=value , PROC=current-procedure
```

that is, local variables are declared and initialized with diagnostic information.

Consider the following example of a procedure FORTC which compiles a FORTRAN program and saves the object module if all goes well; the error procedure COMPERR is used to trap compile errors.

```
1  PROC COMPERR
2  IF ERRCODE>4 RETURN RC,ERRCODE
3  END
4  PROC COPY,TO=FILE,NOTOPTIONAL,FROM=FILE,NOTOPTIONAL
5  CALL COPYPGM,TO=[TO],FROM=[FROM]
6  END
7  PROC FORTC,PROGRAM=FILE,NOTOPTIONAL,PRINT=FILE,
      DEFAULT(*OUTPUT),OBJECT=FILE,DEFAULT(&LOADSET)
```

```
 8   COMPERR,RC : CALL FORTRAN,SYSIN=[PROGRAM],
        SYSLIN=&TEMP,SYSPRINT=[PRINT]
 9   COPY,TO=[OBJECT],FROM=&TEMP
10   END
11   FORTC,PROGRAM=MY.LINSYS OBJECT=MY.LIB(LINSYS)
```

The procedures COMPERR, COPY and FORTC are declared in lines 1 to 10, and then FORTC is called in line 11, with the specification of values for the local variables PROGRAM and OBJECT. The first statement of the procedure FORTC is line 8: the execution of the program FORTRAN with various file definitions. If this program returns a non-zero return code then COMPERR is called with ERRCODE initialized to the return code. If ERRCODE is greater than 4 then control returns to FORTC with the return code indicator still set; this error condition then caused control to be passed back to the calling procedure (line 11) and so on until reaching a level where the error condition is cancelled. If in COMPERR the value of ERRCODE is not greater than 4, the error condition is cancelled, and control passes normally to the next command after line 8. On the occurrence of any other kind of error condition, during the execution of FORTC, control is passed up to the appropriate error routine, thus exiting from lower procedure levels.

SYNTAX OF THE LANGUAGE

A number of data types are recognized syntactically:

CHAR	
WORD	a sequence of letters
NAME	a sequence of letters and digits starting with a letter
ALPHAMER	a sequence of letters and/or digits
FILE	a filename (operating system dependent)
STRING	
INTEGER	
BOOLEAN	
PROC	
ENV	
FUN	

Of these, the first six may be values of variables of type string.

EXPRESSIONS

Variables and constants may be combined to form expressions using the following operators (in order of precedence):

<table>
<tr><td>[]</td><td>variable evaluation</td></tr>
<tr><td>function call</td><td></td></tr>
<tr><td>* /</td><td>multiplication division</td></tr>
<tr><td>+ -</td><td>addition subtraction</td></tr>
<tr><td>< > <= >= = ¬=</td><td>relationals</td></tr>
<tr><td>¬</td><td>logical NOT</td></tr>
<tr><td>&</td><td>logical AND</td></tr>
<tr><td>|</td><td>logical OR</td></tr>
</table>

Brackets are used conventionally, and association is from the left. The only permitted string operators are + (concatenation), = and ¬=.

String constants are normally written enclosed between double quotes. Variables of integer or string type occurring within string constants will be evaluated if enclosed within square brackets.

Examples: A, B, C, D are variables with values 1 (integer), XYZ (string), false (Boolean) and 10 (integer), then we may evaluate expressions as follows:

<table>
<tr><td>(D-A*2)/4</td><td>=</td><td>2</td></tr>
<tr><td>"ABC"</td><td>=</td><td>ABC</td></tr>
<tr><td>"A[B]C"</td><td>=</td><td>AXYZC</td></tr>
<tr><td>A<D|C</td><td>=</td><td>true</td></tr>
<tr><td>A+D</td><td>=</td><td>11</td></tr>
<tr><td>"A"+"D"</td><td>=</td><td>AD</td></tr>
<tr><td>"[A]"+"[D]"</td><td>=</td><td>110</td></tr>
</table>

STATEMENTS

Individual statements may be separated by the character ";". Otherwise they are terminated by "newline" unless the syntax indicates that the statement is incomplete. Individual statements may be grouped into multiple statements by use of command brackets $(and $), and may then be repeatedely or conditionally executed. A label may appear on an otherwise blank line and is preceded by "*". Blank lines are ignored. Comments are introduced by // and terminated by "newline".

variable declaration	LET variable-name = <?| value-expression>
assignment	variable-name := value expression
transfer of control	GOTO label RETURN error-type,error-code EXIT RESULTIS value-expression BREAK LOOP FINISH
conditional	IF Boolean-expression THEN statement UNLESS
repetitive	WHILE Boolean-expression DO statement UNTIL
program execution	CALL program-name,program-environment
procedure call	procedure-expression,arguments,variable-declarations
environment entry	ENTER environment-name
library declaration	LIBRARY filename
library cancel	CANCEL filename
command text inclusion	GET filename

environment declaration	ENV environment-name <a (possibly null) set of declarations of variables, procedures, functions and environments> END
procedure declaration	PROC procedure-name,formal arguments <a (possibly null) set of declarations of statements, which may include RETURN> END
program	<a set of statements>

PROCEDURES AND FUNCTIONS

For the declarations of procedures and functions, the formal arguments consist of a list of items of the form

variable-name = type,option,option,...

separated by commas, where the variable names are those of local variables initialized to the arguments supplied. These values are checked for type at call time. Admissible types are:

char, word, name, alphamer, file, string, integer, bool, proc, fun, env, rest.

The type rest refers to the unanalysed part of the command text treated as a string. Argument options are:

NEVERKEY	argument is positional
ALWAYSKEY	argument is not positional
NOTOPTIONAL	argument must be supplied
DEFAULT(VALUE)	default value for omitted optional argument

Non-positional arguments are keyed using the variable name in the formal argument declaration. An omitted optional argument with no declared default is marked with a special "unset" value, which may be tested by a standard function.

In the procedure call the arguments can be either positional or keyed, and are separated by commas. The variable declarations consist of a list of terms of the form

variable-name = value-expression

separated by commas. To simplify use of the procedure call, string constants wich do not contain special characters do not have to appear enclosed in double quotes. To avoid ambiguity, the evaluation of expressions must now be forced by the use of square brackets.

The syntax of a function call is similar to that of a procedure call. The diferences are:

- the list of arguments and variable declarations is enclosed in brackets, even if it is empty.
- string constants must be enclosed in double quotes, and square brackets are not necessary for forcing the evaluation of expressions.

ENVIRONMENTS

An environment body consists of a set of declarations of procedures, functions and other environments, together with declarations of variables which are common to more than one of the local procedures and functions, and which are not assumed to be declared in an outer environment. An environment is best understood as a set of procedures with similar ends, which use a common set of data.

An example is the job submission environment where the procedures prepare and submit jobs for background execution. The common data would include such items as default job description parameter values, to avoid having to specify these explicitly in each procedure. One such procedure might be for submitting a Fortran job, another for dumping a tape, and so on.

Another example might be the file editing and maintenance environment; yet another might be the environment for compilation and execution of programs in one or more programming languages. Environments may and normally will be nested to facilitate combinations of calls to procedures declared in different environments.

It is to be understood that an implementation will include the provision of several environments, one of which will be the standard environment in which basic functions will be declared.

THE STANDARD ENVIRONMENT

The standard environment will be entered at the beginning of execution of any program in the control language, and it will only be left at the end of the program. In this environment are declared the variables used by the system, which are mainly user settable parameters. Examples are:

- USER the identification of the user (string)
- QUIET suppresses the listing of procedures during execution (Boolean)
- DUMP requests the taking of a dump in case of abend (Boolean)

The functions and procedures declared within this environment include routines for environmental enquiries, type conversions, simple I/O and the standard error procedures. The environmental enquiries include Boolean functions to test whether a name has been declared, if it is "unset", and to test its type. Type conversion functions are provided between integer and string expressions, together with a function which tests if these are possible.

Standard input and output streams are implicitly defined by the system, and will normally be connected to the terminal in the case of interactive use. Simple I/O procedures to output short texts or to input variable values are provided, as is the display procedure which lists all variable and procedure names currently in scope, together with their types and values.

AN EXAMPLE

```
0        0        LIBRARY  SYS.LIB //activates the environment
                           // declarations contained in this file
1        0        ENTER FILEMANIPULATION //activates the
                           //declarations of EDIT and COPY
2        1           LET DOC="MAS.FORT1" //declares and
                           //initializes the variable DOC
3        1           COPY,TO=[DOC],FROM=*IMMEDIATE(%)
                           < Fortran program text >
```

```
 4                  %
                      //copies the input text to file MAS.FORT1
 5         1        EDIT,FROM=[DOC],TO=&TEMP
                      < editing commands >
                      //interactive editing
 6         1        COPY FROM=&TEMP,TO=[DOC]
                      // save the edited text
 7         1        ENTER JOBSUBMISSION
                      // activates the background
                      //job submission procedures
 8         2          FORTGCLG,FROM=[DOC],PARMF=
                         NOSOURCE,JOBNAME=TUESDAY
                      // submits a Fortran job to
                      //compile and execute the program
                      //in MAS.FORT1, changing some
                      //default values
 9         2          DOC:="MAS.FORT2" // assign a
                                 //new value
10         2          ENTER COMPILATION //activates the
                       //foreground compiler procedures
11         3            LET DOC="MAS.FORT3"  //creates
                          // a new variable DOC local
                          // to this environment
12         3            FORTGCLG,FROM=[DOC] //compiles
                          //in foreground the text in
                          // MAS.FORT3
13         3            EXIT //cancels the most recent
                          //environment, including the
                          //local variable DOC
14         2          FORTGCLG,FROM=[DOC] // the program
                          //is read from MAS.FORT2, by a
                          // Fortran job submitted
                          //by this command
15         2          FINISH   //terminate the program
```

The environment FILEMANIPULATION entered at (1) declares the procedures EDIT and COPY. The variable DOC is declared (2) and assigned a string value. In its use as an argument (3) DOC is enclosed in square brackets to force evaluation. The COPY procedure (3) copies the FROM datafile to the TO datafile. In this case, FROM refers to an instream datafile, or immediate document, terminated by "%"

(4). The JOBSUBMISSION environment is entered (7) and includes FORTGCLG, a procedure to submit a FORTRAN job, amongst its local procedures. In calling the procedure (8), PARMF is an argument and JOBNAME a new local variable, since it is the name of a variable declared in the JOBSUMISSION environment, rather than the procedure FORTGCLG. A new value is assigned (9) to the variable DOC declared in (2). The COMPILATION environment is entered (10) and a new variable DOC is declared and initialized within it (11). The procedure FORTGCLG (12) is local to COMPILATION and represents the compilation and execution of the program in the file MAS.FORT3 . On exiting from this environment (13) we return to the scope of the variable DOC declared in (2) and the FORTGCLG procedure declared in the JOBSUBMISSION environment. The control language program is terminated by the command FINISH (15).

CONCLUSION

The above description has highlighted those control language features not normally found in a programming language. Apart from the program execution statement and the use of file names, which are peculiar to a control language, the remaining novel features are a consequence of the desire to serve the interactive user in a well-structured language environment.

The description presented here is necessarily partial, and the author intends to remedy this in the near future with a complete description of the language ARARA/370 designed for use with IBM's OS/MVT or its successors.

No discussion of this kind would be complete without some mention of how the language is to be implemented. For interactive use the translation has to be performed by direct interpretation of the source text. This by no means excludes the possibility of the pre-compilation of procedures and environments into an intermediate code which can be interpreted at execution time. In fact the type checking of arguments would facilitate such a choice. Thus a complete program could consist of a mixture of source text and intermediate code. Such a system is under development by the author at PUC/RJ.

The author wishes to acknowledge the contributions made by Dr M. F. Challis to the development of the ideas expressed here. The work described is partially supported financially by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

REFERENCES

- Barron D. W. and Jackson I. R., The evolution of Job Control Languages, Software - Practice and Experience, 2 (2),143 (1972).

- Cambridge University Computing Service, Cambridge 370/165 Users' Reference Manual (1975).

- Gilman L. and Rose A. J., APL, An Interactive Approach, Wiley, 2nd edition (1974).

- Hazel P., A Common Language for Utilities, internal memorandum, Cambridge University Computing Service (1976).

- Hoare C. A. R., Monitors: An Operating System Structuring Concept, CACM 17 (10),549 (1974).

- IBM System /360 Operating System: Job Control Language Reference, Form no. GC28-6704.