Ц8406

М-30

2368/2-77

СООБЩЕНИЯ
ОБЪЕДИНЕННОГО
ИНСТИТУТА
ЯДЕРНЫХ
ИССЛЕДОВАНИЙ

ДУБНА

20/VI-77

E10 - 10544

D.C.Marinescu

CDL AS A SYSTEM IMPLEMENTATION LANGUAGE IN SCIENTIFIC ENVIRONMENT

Part I

1977

E10 - 10544

D.C.Marinescu

CDL AS A SYSTEM IMPLEMENTATION LANGUAGE IN SCIENTIFIC ENVIRONMENT.

Part I

Маринеску Д.К. **E10 - 10544**

Использование CDL в качестве языка описания и внедрения систем математического обеспечения в научных целях. Ч.I

В работе характеризуется специальный язык, который удобен для описания системных программ: трансляторов, программ - редакторов, мониторов и т.п. Приводится описание его основных синтаксических конструкций и особенностей использования. Транслятор с этого языка внедрен автором работы на ЭВМ CDC-6500 ОИЯИ, и в работе описаны способы обращения к данному транслятору и возможности использования языка на машинах других типов.

Работа выполнена в Лаборатории вычислительной техники и автоматизации ОИЯИ.

Сообщение Объединенного института ядерных исследований. Дубна 1977

Marinescu D.C. **E10 - 10544**

CDL as a System Implementation Language in Scientific Environment. Part I

The special language is characterized which is convenient for system program description: compilers, editors, monitors, supervisors,etc. The definition for its basic syntactical constructions and some notes about the possible usage are described. The version of the compiler for this language has been implemented by the author for the CDC-6500 at JINR, and this report contains the description for the access to this compiler and some possibilities of using this language on different computers.

The investigation has been performed at the Laboratory of Computing Techniques and Automation, JINR.

Communication of the Joint Institute for Nuclear Research. Dubna 1977

Introduction

It is a proven fact that scientific communities require special software to be designed to fit their demanding needs.

Such communities are very sensitive to what it is generally called "user convenience" and this is reflected into a considerable effort to bring the machine closer to user wishes. Also a wide spectrum of machines are generally in use and it is always a problem to make the facilities, available on one machine, work on the others.

Consequently a lot of work is done, under completion or planned to be done in such areas as programming language and operating system development practically at every research center with computing facilities.

A system implementation language is an extremely useful tool which can cut down drastically the expenses related to a new system project and can reduce the time needed for its completion.

CDL (Compiler Description Language) is a high level which can be profitably used as a system implementation language; compilers for it are available on most machines currently used in scientific communities (CDC, IBM, PDP, etc.). It is well suited for writing system programs (compilers, utility programs, editors, and even monitors). Even such sophisticated systems like REDUCE will be in the near future written in CDL. A great advantage comes from the fact that the number of people using CDL is growing faster.

CDL has been developed (at the Technical University of West Berlin by Professor Koster) primarily as a compiler writing system; now it is undergoing revision and a new version CDL2 will be available soon.

1. CDL extension on syntax directed compiling techniques

There is an intimate relation between a programming language and a compiler for that language; that means that as soon as we are able to construct a formal description of a programming language we should be able to provide the formal definition of a compiler for the language. Such an idea is very attractive since from such a formal definition we can dream some means to automatize the compiler built-up.

But as we know the two mechanisms, the language syntax and its semantics, cannot be separated from each other when defining a compiler for thelanguage;the syntax enables us to recognize all the sequences of characters which are valid sentences in the language and to derive parsing trees for them while the semantics attaches a meaning to a parsing tree of a correct program.

The formalization of the syntax is well mastered today via syntax directed techniques but there is no formalization of the semantics. As a result, manual techniques are the only solution to consider when semantic actions are to be included into the compiler.

Compiler Description Language aims to extend the use of syntax directed techniques so that all but the very kernel of the semantics of a compiler can be defined by syntax and only that kernel must by defined by other means.

Thus, a compiler written in CDL is machine independent but for a small set of primitive actions implemented as macros in the machine language of the host computer.

Koster /1, 2/ provided a CF grammar (Context Free) with several extensions:

- affixes as a mechanism for parameter communication to and from a procedure;
- actions embedded into syntax;

- jumps and labels;
- global variables.

The result is a beautiful high level programming language well suited not only for compiler writing but for the most of the problems a system programmer is faced with.

2. CDL specifications

There are four types of symbols accepted as the building stones of CDL:

a. The tags; they fill the name space of a CDL program. A tag consists of a letter, possibly followed by a number of letters of digits between which spaces are ignored.

b. The constants; a constant is a sequence of digits between which spaces are ignored.

c. The special symbols; + - *; (=) , :

d. The bold symbols; they are defined as strings of characters other than accents, enclosed between accents. There are several reserved bold symbols used to:

d1. establish the type of a user defined tag;

d2. name the parameters of a procedure and the result;

d3. define a command for the compiler.

Each tag (name) has associated with it three attributes:
- the type attribute;
- the state attribute;
- the soft attribute.

These attributes are specified using the reserved bold symbols of class d1 (see table 1).

Table 1. The reserved bold symbols.

class d1	class d2	class d3
'external'	'restore'	'1'
'action'	'unrestore'	'2'
'pointer'	'trace'	'3'
'flag'	'untrace'	'4'
'macro'	'long'	'5'
'list'	'short'	'result'

The type attribute associates an object with a given name with one of the six classes of objects which are handled by CDL: flags, actions, predicates, pointers, lists, labels.
The flags are used to test bivalent conditions; for example the flag EQUAL+X+Y which tests if its first parameter X is equal to the second one, Y, can be defined as:

```
            'MACRO'  'FLAG'  EQUAL
            EQUAL='1'='2'.
```

The actions and the predicates are the two types of procedures available in the language. While a predicate can return either a true or a false value, an action always returns a true value. Both of them can have parameters and local variables; as an example the action INCR+X which adds 1 to its only argument can be defined:

```
            'MACRO'  'ACTION'  INCR.
            INCR='1':='1'+1.
```

Now we will define the predicate ADD ONE+X+Y which tests if its parameters are equal and if so, increments the first one:

```
            'PREDICATE'  ADD ONE.
            ADD ONE+X+Y:
                EQUAL+X+Y, INCR+X.
```

The pointers and the lists are the only data structures available in CDL; the pointer corresponds to the scalar and the list to the linear array of data. In the following example the pointer ALPHA is defined and initialised with the value 1200 and the array BETA of 1200 elements is defined:

```
            'MACRO'  'POINTER'  ALPHA=1200.
                     'LIST'  BETA(1:ALPHA).
```

The labels are used to identify the beginning of a sequence; jumps to labels provide the means to execute repeatedly a sequence of operations.

The second attribute of a tag is the state attribute; the state of a tag can be: defined or rapid.

The third attribute of a tag is the sort attribute; it defines the domain of a tag. There are three sorts:local,global and macro.

Table 2 presents a summary of the attributes which correspond to certain specifications.

Table 2

Tags attributes and their specification

Tag specifications	State	Sort	Type
'external' 'action'	defined	global	action
'external' 'predicate'	defined	global	predicate
'external' 'pointer'	defined	global	pointer
'external' 'flag'	defined	global	flag
'action'	applied	global	action
'predicate'	applied	global	predicate
'macro' 'action'	defined	macro	action
'macro' 'predicate'	defined	macro	predicate
'macro' 'flag'	defined	macro	flag
'macro' 'pointer'	defined	macro	pointer

As far as the structure of a CDL program is concerned there are six types of statements: specifications, declarations, commands, comments, rules and starting symbol statements.

The specifications and the declarations are used to define the attributes of a tag. It should be noted, that there are default specifications; for example for a procedure an internal specification 'predicate' is always assumed so that whenever we want to define an 'action' we must specify it explicitly.

The commands are grouped together into three pairs:

```
'restore', 'unrestore' (default is 'unrestore')
'long'   , 'short'     (default is 'short')
'trace'  , 'untrace'   (default is 'untrace').
```

The commands are used to establish a certain mode for the compiler-compiler itself. The first pair defines the parsing mode; the restore parsing allows a more efficient backtracking. The second pair controls the output of the compiler-compiler; the short mode suppresses the layout characters embedded into the output strings. Tracing mode determines the printing on the output of all attributes of all locals and globals.

The comments are included into the CDL program for the sake of readability.

The rules are the executable statements of the CDL program. Each rule consists of a set of alternatives separated by semicolons (;) and the last alternative is terminated by a period (.). In turn each alternative is a sequence of members separated by commas (,). When a rule is executed the alternatives are tried in the order of occurrence; if one alternative succeeds, the whole rule succeeds and is thus terminated. If it fails, the next alternative is tried until eventually one of them succeeds or if none of them does it, the rule itself fails. All the members of an alternative must succeed in order to have a successfull alternative. A good way of representating intuitively the flow of control in a CDL program is the following: we write the alternatives of a rule on a single line (whatever long that line might turn out to be), one alternative under the other. When successful, the control goes on the horizontal line from left to right until the semicolon is encountered but when unsuccessful the control descends on a vertical line. CDL has a block structure, each rule being a block which can communicate with the others via its parameters; the parameters are either called by name or by reference. There are also local variables of a procedure.

The starting symbol defines the action which has to be executed to produce the desired result.

3. Programming in CDL .

We want to present here examples of small CDL programs and to show the organization of a realistic compiler written in

CDL, in order to get some insight into the nice features of this programming language suitable for system program design.

A CDL program runs in a certain environment created with the aid of a library of macros and functions; we sould not be mislead by the striking simplicity of the CDL program itself since a lot is going behind the sence.

First we present a program which reads a string of characters and prints it in reverse order.

```
'MACRO''POINTER' NLCR CODE=110.
'EXTERNAL''ACTION' RESYM,PRSYM.
'ACTION' COPY REVERSE.
     COPYREVERSE - CHAR :
        RESYM + CHAR , EQUAL + CHAR + NLCR CODE ;
        COPY REVERSE , PRSYM + CHAR .
'RESULT' COPY  REVERSE.
```

Here the environment will contain the read and print symbol routines, the definition of the character NLCR, the EQUAL macro. The last program statement informs that the result is expected from the execution of the COPY REVERSE procedure (in fact, the only procedure of this program). The action COPY REVERSE has two alternatives; the first one reads a character from the input string and fails unless this character is a NLCR. Control is thus passed to the second alternative which implies a reccursive call of the action COPY REVERSE and when this call is successful the character is printed. So that the characters in the input string are read and stacked and only when the NLCR character is found, then the first time the procedure is successful and starts printing from the top (last character read) to the bottom (first character read) of the stack. This procedure has a local variable CHAR which appears as a parameter for the external actions RESYM, PRSYM and for the EQUAL predicate.

We shall now present the organization of a real life compiler written in CDL (examples are taken from the CDL to ALGOL 60 compiler written for CDC 6500 by J.Jackel). The compiler is organized into four sections. The first section creates the running environment. Here there are specified :

- the commands establishing the compiler mode;
- the external procedures used mainly for I/O;
- the definition of different characters used by the language to be translated (CDL);
- several flags which test if digit, letter, specification, etc.;
- the basic macros for elementary operations;
- global variables.

Samples from the actual code of this section are given below.

```
'LONG'
'RESTORE'
'EXTERNAL''ACTION' RESYM, PRSYM,EXIT,NEW PAGE.
'MACRO''POINTER' SPACE CHAR=105,TAB CHAR=109,NLCR CHAR=110,
    ACCENT CHAR=104,QUOTE CHAR=87,NULL CHAR=54.
'MACRO''FLAG'
    WAS LETTER=('1' > 27)'"AND"'('1'<54),
    WAS DIGIT =('1' >53)'"AND"'('1'< 64),
    WAS LETDIG=('1' >27)'"AND"'('1'< 64),
    WAS SPECIFICATION=('1'>63)'"AND"'('1'<104) .
'MACRO''ACTION'
    MAKE '1':='2' ,
    DECR='1':='1'-1,
    ADD ='3':='2'+'1' ,
    SUB ='3':='2'-'1' ,
    DIVREM = '3':='1''"/"' '2'; '4':='1'-'2' *'3' ,
    MARK='1':=-'1' .
```

The second section is devoted to the input and output processing. The main action to perform the printing is:

```
'ACTION' OUT.
    OUT +X -Y :
       WAS TAG +X, PRINT TAG +X ;
       WAS BOLD+X, PRINT BOLD+X ;
       WAS SPECIFICATION+X,PRINT SPECIFICATION+X ;
       WAS CONSTANT +X,GET CONSTANT+X+Y,OUTINT+Y;
       OUTINT +X.
```

Here the four flags (WAS ...symbol type) identify the parameter X as a certain type of symbol (tag,bold,specification,constant) and directs the printing of it. OUTINT is an action to print an integer and uses several other actions which will also be defined:

```
'ACTION' OUTINT,OUTINT1,SPACE,SPACES,NLCR.
OUTINT +X -QUOT -REM :
     SPACE,
     LESS+X+0,MAKE+REM+X,MARK+REM,PRCHAR+MINUSCHAR,OUTINT1+REM;
     EQUAL+X+0,PRCHAR+NULLCHAR,SPACE;
     DIVREM+X+10+QUOT+REM,OUTINT1+QUOT,ADD+REM+54+REM,PRCHAR+REM,
                                                            SPACE.
OUTINT1+X-QUOT-REM:
     EQUAL+X+0;
     DIVREM+X+10+QUOT+REM,OUTINT1+QUOT,ADD+REM+54+REM,PRCHAR+REM.
SPACE:
     PRCHAR+SPACECHAR.
SPACES+X-N:
     MAKE+N+0,
     START: LESS+N+X,SPACE,INCR+N, :START.
```

Here is the first time we use a label (START:) and a branch to it, (:START). The action SPACES outputs a number X of space characters.

```
PRCHAR+X:
     EQUAL+X+NLCRCHAR,NLCR;
     EQUAL+X+TAB CHAR,SPACES+8;
     EQUAL+X+NIX;
     PRSYM+X,INCR+POS.
```

Here it should be noted, that POS is a global variable which defines the position on the output line; it is incremented when a character is printed and reseted when a new line should start.

```
NLCR:
    PRSYM+NLCR CHAR,MAKE+POS+O.
```

The third section of the complier deals with the recognition and translation of CDL rules.

The fourth section is the nucleus of the compiler.

```
'ACTION' SENTENCE,START SYSTEM,COMPILER DESCRIPTION.
SENTENCE:
    START SYSTEM,BLANK LINE,NEW LINE,PRINT HEADER+START,
    COMPILER DESCRIPTION,NEW LINE,PRINT HEADER+END,BLANK LINE,
    NEW PAGE,POST MORTEM.
COMPILER DESCRIPTION:
    START: SPECIFICATION,  :START;
           DECLARATION, :START;
           COMMAND, :START;
           COMMENT, :START;
           STARTING SYMBOL;
           RULE, :START;
           SKIP UNTIL POINT, :START.
'RESULT' SENTENCE.
```

4. The flow of control in a CDL program

The output of the CDL to COMPASS compiler-compiler is a set of COMPASS pseudo-instructions and macro calls. We present in table 3 the expansion of one of the programs listed in the previous section.

Table 3

COMPASS expansicn of a CDL program

STM	CDL statement		COMPASS statement	
			IDENT	CDL
			XTEXT	CDLTEXT
		GET	MACRO	P1,P2
		PUT	MACRO	P1,P2
			ZINIT	
155	'MACRO''POINTER'NLCRCODE=110			
156	'EXTERNAL''ACTION'RESYM,PRSYM		EXT	RESYM
			EXT	PRSYM
157	'ACTION' COPY REVERSE.		ZBLOPEN	(GO),(0),(4),(158)
158	COPY REVERSE - CHAR :		ZLOFREE	(1),(3)
159	RESYM+CHAR,		ZCALL	(RESYM),(159)
160	EQUAL+CHAR+NLCRCODE;		EQUAL	(B4+3),(=110)
			ZNEGJUMP	(1),(GO)
			ZJUMP	(999),(GO)
161	COPYREVERSE,PRSYM+CHAR.		ZLABDECL	(1),(GO)
			ZCALL	(GO),(161)
			ZLOFREE	(1),(3)
			ZCALL	(PRSYM),(161)
			ZLABDECL	(999),(GO)
			ZRETURN	
162	'RESULT' COPYREVERSE		ZENTRY	
			ZSTCALL	(GO),(162)

We can see in table 3, that the pseudo-instruction XTEXT informs the COMPASS assembler that the deck named CDLTEXT (from the file OLDPL, since no X parameter is to be specified on the COMPASS control card-see next section for the program deck-) is the source which contains the text of the macros listed below this instruction. New versions of the macros GET and PUT have been defined ; during the COMPASS assembly two warning messages inform the user, that new versions of these two macros are overriding the old ones.

The control goes first to the macro INIT and the message 'START OF THE CDL PROGRAM' acknowledges the beginning of the program execution; then the control is passed to the RUN TIME ROUTINES (entry point CDLSYS) when base register initialization takes place. The macro ENTRYPOINT activates the TERMINALS when requests like INITIALIZE FOR READING occur. The macro CALL provides linkage to different externally defined actions as requested by the user. Finally, the macro STARTCALL provides the necessary action for the 'RESULT' rule in the user program. The return in the RUN TIMF ROUTINES (to the entry point CLOSE) and the message 'END OF CDL PROGRAM' are associated with the termination of the program execution.

5. The availability of CDL

There are two high level versions of the CDL compiler-compiler; one is written in ALGOL 60 and the other in CDL. Also a low level version for the CDC 6500 computers, written in COMPASS, is available.

We have installed here at Dubna on the CDC 6400 machine only the low level version since we believe that it is far more efficient that the other.

It should be pointed out that CDL compiler-compilers are available on a variety of computers : IBM 360 and 370, RIAD, CDC - 6000, PDP-11, etc. The bootstrapping technique can be used to construct CDL compiler-compilers for practically every available computer; with a reasonable effort such a compiler has been designed for the R 10 computer /4/.

The great advantage of using CDL when writing system programs (compilers, editors,librarians,utility programs, etc.) results from its power to describe well structured collections of objects. For example the CDL to ALGOL 60 compiler-compiler contains less than 200 rules; the readability of such a program is tremendous as compared with an assembler written program performing the same functions.

On the other hand, a CDL written system program can always be tested on a powerfull machine and only when successfully runs, there it can be installed on any of the machines provided with a CDL compiler-compiler.

The suggested set-up for the job when a compilation and an execution of a CDL written program is desired is:

```
JOBCARD,CM55000,T200.
ATTACH,CDLSYS,ID=COMP,MR=1.
ATTACH,OLDPL,CCLIB,ID=COMP,MR=1..
ATTACH,ANDY1,ID=COMP,MR=1.
UPDATE,Q,D,C=CARDIN.
MAP,ON.
LOAD,ANDY1,CDLSYS.
EXECUTE,,I=CARDIN,C=CARDOUT.
REWIND,LGO,CARDOUT.
LOAD,LGO,CDLSYS.
EXECUTE.
end of record
*COMPILE PRE,MACLIB
end of record
*ADDFILE
*DECK PROG
---------------------------------------------------
    The deck containing the program written in CDL
---------------------------------------------------
end of record
    -------------------------------
    The data for the CDL program
    -------------------------------
end of file
```

Three permanent files CCLIB,CDLSYS and ANDY1 are used. The file named ANDY1 contains the compiler-compiler itself in the loadable format. The CDLSYS file contains the CDL RUN TIME ROUTINES. Both files must be loaded into the LGO file to create the running environment for the compilation step (CDL to COMPASS); this is executed as requested by the following control cards:

```
 LOAD,ANDY1,CDLSYS.
EXECUTE,,I=CARDIN,C=CARDOUT.
```

The input file for the previous step (CARDIN) is created using the UPDATE program:

```
UPDATE,Q,D,C=CARDIN.
end of record
*COMPILE PRE,MACLIB
end of record
*ADD FILE
*DECK PROG
    -------------
    CDL program
    -------------
end of record
```

The new library created by the UPDATE program (the file named CARDIN) will contain: the deck PRE (the PRELUDE), the deck MACLIB (this contains a library of procedures used by the CDL program), the deck PROG (the user CDL program).

As a result of the CDL to COMPASS translation the file CARDOUT is produced. The following control cards will direct the COMPASS assembly, the loading and the execution of the program:

```
COMPASS,I=CARDOUT.
LOAD,LGO,CDLSYS.
EXECUTE.
```

A final point is that in our version the following characters are to be used:

- the colon (:) is to be replaced by the percentage character (%) with the BCD code 16 and the punching code 8-6.
- the apostrophe (') used to define the bold symbols has BCD code 55, punch code 11-8-5 and will be printed as ↑ character.

References

1. Koster C.H.A. A compiler-compiler MCA-Report, 121, 1971.
2. Koster C.H.A. Using the compiler-compiler. Chapter 4 in Compiler Construction; Springer 1974.
3. Jackel Joachim C. Bootstrap Eines CDL-Compiler Auf Die CDC 6500 Technische Universität Berlin, April 1975.
4. Jackel Joachim C. A CDL compiler-compiler for the R 10 mini-computer. In Minicomputer Forum 1975.

Received by Publishing Department
on March 30, 1977.