PUC

Series: Monographs in Computer Science
and Computer Applications

Nº 8/70

THE COMCOM-SOFTWARE WRITING SYSTEM

by

Arndt von Staa

Computer Science Department - Rio Datacenter

CENTRO TÉCNICO CIENTÍFICO
Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

ABSTRACT:

The COMpiler COMpiler system was primarily designed as a tool for compiler writing. By compiling we mean any kind of automatic artificial language translation. In general the system, as seen by the user, is composed of two very flexible languages; a syntactic metalanguage and a semantic metalanguage. We required the object compiler produced by this system to be efficient and not just another academic research tool. Powerful capabilities for compiler writers were supplied that enable the design of efficient object compilers which produce good target codes. Moreover the system also provides for experimental work on compiling techniques, leading to comparative study of different compiling strategies.

A feature a system of this kind ought to have is as much machine independence as possible. For that the COMCOM system compiles object compilers into a pseudo code specially designed for writing compilers. The translation of the intermediate code to existing machine languages can be done by assembler programs specially written for each of the actual object machines. The machine independence for the target language is obtained by leaving to the user the choice of which is going to be the output code.

Since a compiler should also serve various users with only rudimentary knowledge of the languages it must be able to give a large amount of precise error messages. For that the system provides in its syntactic metalanguage not only features for recognition but also for error detection

./.

The COMCOM system was also designed as an aid to research in both formal and natural languages. Because of this many different forms exist for describing the sentences of the syntactic metalanguage with corresponding straightforward semantic high level facilities.

A certain context sensitiveness is allowed by the possibility of runtime modification of the sentences of the syntactic metalanguage of COMCOM. These modifications refer to: conditional recognition, different types of backtracking, insertions and deletions, use of parameters within the pattern description, embedding of executable statements.

1 - STRUCTURE OF THE SYSTEM:

The System Components are the Following:

A - Source language of the object compiler or of any set of transformation rules. This source language may be any of a wide class of languages, such as: general purpose programming languages, problem oriented languages, preprocessors etc.

B - Target language (language that will be produced by the object compiler). The language B, may also be any of a large class of languages. In general: high level language, an assembly language, or machine code. In this text it is supposed that B is accepted by some processor.

L - COMCOM language in which the object compiler (A/B) is defined (syntactic and semantic metalanguages).

M - Machine language in which the COMCOM compiler (1/L) is written.

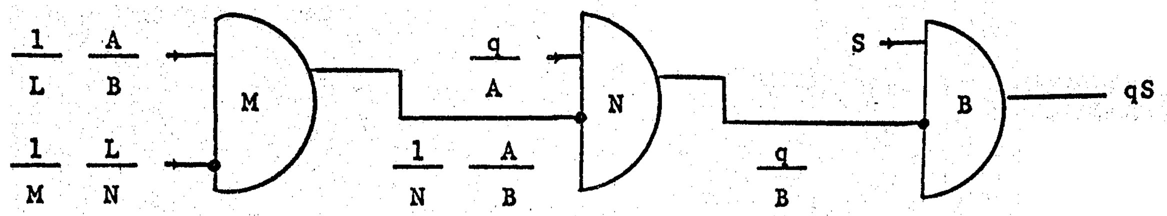
N - Intermediate code generated from L. The execution of this code is done either interpretively or the code is assembled into an existing machine language (not necessarily of the same computer).

q - A set of sentences written in A.

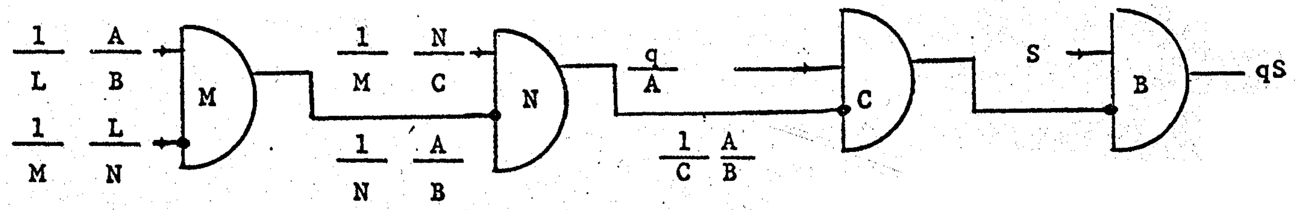
S - Data for q.

A very convenient notation to represent the interaction among these components can be displayed using the notation give in [4].

./.



If the transformation $\left(\frac{1}{N} \quad \frac{A}{B}\right)$ is to be done directly rather than interpretively, the previous scheme becomes



Where A, B, L, M, N, q and S have the same meaning as before, and C is the machine language of the machine where the object compiler (A/B) will run.

2 - AN ABRIGED DESCRIPTION OF THE COMCOM LANGUAGE:

2.1 - Block types

The COMCOM language (L) has a blocked structure. The following block types exist:

External block	- comprises all statements of the program.
FUNCTION blocks	- functions or subroutines which will be executed either by the appearence of the function's name or by a CALL statement. All functions or subroutines in the COMCOM language are recursive. The functions or subroutines may have 0 or more arguments.
MACRO blocks	- a MACRO is a compound structure containing a syntactic part ("name") and a semantic part. The "name" of a MACRO block is a pattern description. The semantic part is a set of sentences to be activated in case an input string matches the name. There may be several MACRO blocks of the same level within the same block and this may be carried to any depth. The input phase is initialized by a COMPILE statement. The action of a COMPILE statement is to pass control to the pattern description list (sentence) of the first MACRO of a given level. If no match succeeds, then the next MACRO sentence of the same level is tried. This process continues until either a match succeeds, or the list of MACRO sentences is exhausted. In the

./.

latter case, the first unit of the input is dropped and the matching phase is restarted with the first MACRO sentence of this level. If the dropped unit was the end of file mark of the input string, then control is passed to the first instruction after the COMPILE statement.

When a match succeeds, then the semantic part of the MACRO block is entered. When returning from the MACRO block after execution, all input units belonging to the input string are dropped from the input and control is passed to the instruction following the COMPILE statement.

BEGIN blocks - blocks which will be executed when the sequential flow of execution passes control to the first instruction of the BEGIN block. BEGIN blocks may be used only within pattern descriptions. The BEGIN block is terminated by an END statement. The BEGIN and END statements may be regarded as a special type of parenthesis enclosing one or more statements. All units read until the initialization of a BEGIN block are available for the user within the block. After the END statement of a BEGIN block the pattern description regains control.

Within blocks any statements may be written, including such statements as COMPILE, MACRO, FUNCTION, etc. All blocks are ended by an END statement.

For FUNCTION, MACRO and the external block, the END statement also corresponds to a RETURN statement.

./.

2.2 - Data Types

The following data types are handled by the COMCOM language:

FIXED	- fixed point (binary) arithmetic data.
CHARACTER or CHAR	- character string data. The strings may be 0 up to 300 characters long.
BIT	- bit string data. The strings may be 0 up to 300 bits long.
LABEL	- label or program reference data. Label constants, arrays and expressions are allowed.
UNIVAR	- syntactic unit data. This data type may be used for matching input strings or it may appear in arithmetic expressions or in assignment statements.
KEY	- pattern description data. This data type is used to gain access to input units or to describe pattern prototypes. The use of the KEY statement and KEY data type will be seen later.

The conversion from one data type to another is done automatically by the system.

2.3 - Pattern units in pattern descriptions:

Input units are matched one by one with the units to be recognized in the pattern description indicated by a MACRO name. If the unit

./.

available at the input satisfies what has to be recognized, then the execution proceeds.

If they are not equal, then an alternative pattern description is tried.

The input will be moved back (backtracked) a sufficient number of units to ensure that all input units analyzed in the current failed pattern are tried again. If no more alternative patterns exist, then the next MACRO is tried.

The existing pattern units are:

Unit constants -	elements which have to yield an equal match to the input. Ex: " + ", "DECLARE" , "LITERAL".
Unit variables -	variables of the type UNIVAR. The input has to yield an equal match to the current value of the variable.
Predefined units -	are categories of pattern elements. The units of the system are:

$I	- any identifier. Ex: A, ABC, X1234YZ, il.
$C	- any constants . Ex: 1, 1234567890.
$L	- any literal. Ex: 'ABCD','+', '1234'
$IC	- any identifier or constant. Ex: A, 123, ABC, 456.
$AOP	- any arithmetic operator. + - * / **.
$COP	- any comparison operator. LT GT LE GE NL NG EQ NE.
$U	- any syntactic unit. Ex: ABC,'ABC', 123, LT, -, NOT

./.

2.4 - General Statements:

The COMCOM language contains general statements like:

- DECLARE or DCL
- assignment statement
- IF bit-expr THEN stat ELSE stat
- DO index = expr TO expr BY expr WHILE bit-expr
- GO TO label constant or label expression.

Since most of the statements above follow the rules of PL/1, I will not go into detail describing them in this text [6].

The output of messages and code is provided by the statements OUTPUT and OUTCOD respectively. The output is format free, and the items to be output are concatenated leaving no intervening blanks. The output statement outputs messages which are intermingled with the card images of the input. The OUTCOD statement outputs on an intermediate file, thereby allowing the use of the produced code for later applications.

2.5 - Internal Functions:

Several internal functions are included. These functions aid in:

- character string manipulation,
- special actions during the matching phase,
- symbol table handling,
- special access to input units,
- output editing.

Some other features will be illustrated in the examples of the following section.

./.

3 - EXAMPLES OF PATTERN DESCRIPTION:

Well formed forms are described through MACRO statements. User defined prototypes, KEYs, may also be used of describing patterns. Access to input units is gained in arithmetic expressions by using KEY names in the expressions.

Notice that predefined units, e.g. $I, $U, etc., are considered as being system defined KEYs. If the scope of a given KEY admits more than one input unit, then only the first of these units will be accessible.

Example 1: Let G be a grammar which produces the following sentences:

A, AA, AAA, ... , AA ... A $\{A\}^+$

The productions of this grammar could be recognized by the following KEY:

```
KEY   G = "A"  0-("A").,
```

Where:

G —— name of the KEY
"A" — unit constant
0 - ( ) - declares that the pattern description inside the parenthesis is to be repeated 0 or more times.

When using this KEY for recognition, one will never be able to know how many times an A was found at the input since only the first input unit of the key is accessible.

The scope of the KEY would be $\underbrace{A\,A\,\ldots\,A}_{G}$

./.

Another recognizer would be:

```
KEY   GB = "A".,
KEY   GA = 1 - (GB).,
```

This scheme allows us to know how many A's occurred, simply by conting the occurrences of KEY GB.
The scope of the KEYs would be $\underbrace{\underbrace{A}_{GB_1}\ \underbrace{A}_{GB_2}\ \dots\ \underbrace{A}_{GB_n}}_{GA}$

A third solution could be:

```
KEY   GA = ("A"  GA  OR  "A").,
```

the scope of the keys would be $\underbrace{A\ \underbrace{A\ \dots\ \underbrace{A}_{GA_n}}_{GA_3}}_{GA_2}$

This KEY also allows us to know how many A's have been read, the number of occurrences of KEY GA being equal to how many A's have been read.

The definition above is recursive. When an A is found at the input, the KEY GA will be retried until no more A's exist at the input. In this case the last call to GA fails, and, consequently, the first alternative pattern "A" GA fails. Therefore the last occurrence of A is looked up

./.

twice at the input. To avoid this matching duplication, the pattern could be defined in the following way:

```
KEY  GA = "A"  (GA  OR).,
```

In this case the second alternative is null, so if GA fails the second alternative cannot fail and consequently the recognition will succeed.

<u>Example 2:</u> Let G be a grammar which produces sentences of the form $A^n B^n C^n$, $n > 0$. Sentences of this grammar could be recognized by:

```
KEY D = 1 - ("A" SET INDEX TO I) "B" I("B") "C" I ("C") .,
```

The statement SET INDEX TO assigns the current value of the repetition counter to the fixed variable I. Since the repetition counter is only incremented when the list to be repeated is exhausted, when the repetition ends the value of I will be one unit less than the final value of the repetition counter.

The net effect of this KEY is: the input is compared with A until no more A's exist at the input. After that the input is compared exactly I+1 times with B "B" I("B") .

If this is true then the input is compared exactly I+1 times with C. Of course, if more than I+1 C's exist at the input, the remaining ones are left for later recognition. Sentences should be defined having a final delimiter, e.g. ".,".

./.

Example 3: Let us consider the DIMENSION statement, supposing that '.,' is the final delimiter of the statement.

```
Ex: DIMENSION  A(10), B(2, 3), C(5, 6, 7).,
```

This string could be recognized by the following statements:

```
KEY  DIMUNIT = $I "(" $C 0-2 ("," $C) ")".,
KEY  DIMLIST = DIMUNIT 0-("," DIMUNIT).,
   .
   .
   .
MACRO  "DIMENSION" DIMLIST ".,".,
```

The MACRO declaration indicated is the name declaration of the function which will handle DIMENSION statements. It is easy to see from the example that the KEY DIMLIST could be used in other declarative statements like INTEGER, REAL etc. The example shown above will not recognize statements like: DIMENSION A(10) B(5)., because of the absence of the comma. The KEY DIMLIST could be rewritten to allow this kind of error to be detected:

```
KEY DIMLIST = DIMUNIT 0-(("," "OR"., BACK UNIT 1 OR BEGIN.,
OUTPUT 'MISSING A  ",".' ., END.,) DIMUNIT).,
```

If the alternative pattern ".," were not included, then an error would be signaled at the end of each DIMENSION statement. The statement BACK UNIT n (1) will again make available the n (1) last units of the input string. In the case shown, one unit, the last one, will again be available at the input. The effect is that the units backtracked will be read twice. The

./.

BEGIN block is initialized when both "," and ".," fail to be at the input. We can see that the BEGIN corresponds to a remaining part of the description (DIMUNIT) corresponds to the termination of the recognition.

<u>Example 4:</u> Consider the grammar which produces the following set of strings: $\{AB, BA\}^+$. In this grammar the order of occurrence of A and B in string AB is immaterial.
A recognizer could be:

```
KEY  AB = UNORD("A" "B").
KEY  SET1 = 1 -(AB).,
```

the UNORD statement allows patterns where the order of occurrence is disregarded. The UNORD statement can also handle null patterns.

```
Ex: KEY  AC = UNORD("A" 0-1 ("X") 2 ("Y")).,
```

would recognize the following strings:

AYY, YYA, AXYY, AYYX, XAYY, XYYA, YYAX, YYXA

the description 0-1 ("X") might be empty, when a possible null sub-pattern is described.

The null condition will only be recognized after a match of each sub-description of an UNORD list has been tried.

Several other pattern description statements exist in the syntactic metalanguage part of the COMCOM language, a few more of which will be listed below:

./.

RECOGNIZE unit-variable - matches input with the variable unit-variable which may be subscripted.

BACK KEY key-name-clears all previous occurrences of KEYs up to and including the last occurrence of the KEY key-name. The input is not affected.

BACK READn- sets the last n input units to the null string; it destroys information contained in the last n input units.

FOUND key-name unit-value-inserts an occurrence of KEY key-name and its corresponding input value unit-value. Unit-value may be unit constant or unit variable.

UNMARK, MARK-are statements which change the status of the registering process. UNMARK inhibits the registering; it is resumed if MARK is encountered.

Labels - labels are allowed to be used within pattern descriptions. They may be referenced by GO TO statements thus allowing transfer of contrd within the pattern description.

4 - EXAMPLE OF A SIMPLE COMPILER:

The compiler to be used as an example will only be able to compile assignment statements, where the expression contains variables, constants and the operations + and —.
Neither symbol table handling is done nor priority of operations is considered.

./.

```
      KEY AEXP= TERM 0-(AOP TERM).,
      KEY TERM= ($I OR $C).,
      KEY AOP = (" + "  OR  " - ").
F1..  FUNCTION OPND(A KEY) CHAR.,
      IF A.$I EQ '' THEN RETURN '=' CAT A.,
                    ELSE RETURN A.,
      END F1.,

M1..  MACRO $I ''=''  AEXP ''.,''.,
      OUTPUT '    CLA    ', OPND(TERM).,
      DO I=1 BY 1 WHILE AOP(I) NE ''.,
         IF '+' EQ AOP(I) THEN OUTPUT '     ADD     ', OPND(TERM(I+1)).,
                          ELSE OUTPUT '     SUB     ', OPND(TERM(I+1)).,
      END.,
      OUTPUT '     STO   ', $I.,
      END M1.,

M2..  MACRO ''  '''EOF''' ''.,STOP., END M2.,

P1..  COMPILE., GO TO P1.,
      END COMCOM.,
```

The function OPND is of type CHAR. The string length of functions is always varying and must not be declared. The argument of the function is of type KEY, thus allowing us to again access to defined input units. The subexpression A. $I has the following meaning: obtain the occurrence of $I within the occurrence of A. If this occurrence is non-existente, it corresponds to a null string. Therefore A will always correspond to an occurrence of TERM in the input. If A.$I is null, then A. $C cannot be null, and TERM is a constant. The subexpression TERM (I+1) gets access to the I+1th occurrence of the KEY TERM. If the I+1th occurrence does not exist, I+1 is greater than the total number of occurrences, then the value obtained by TERM(I+1) will be a null string.

The unit constant " "'EOF''' " corresponds to the end of file mark. The main program

```
P1 .. COMPILE., GO TO P1.,
```

will loop until all assignment statements have been compiled. The loop will be discontinued by the second MACRO.

These examples were run in the IBM 7044/1401 system at the Pontificia Universidade Católica (PUC) of Rio de Janeiro.

5 - POSSIBILITIES OF EXPANSION:

Several extensions could be made on the system to enhance its flexibility:

1- information retrieval functions which would allow it to match input againsts large files or dictionaries.

2- functions which allow the linkage to I/0 and supervisory subroutines

3- file handling and editing functions which would allow more flexibility of input and output.

4- partitibn declarations which would make possible the subdivision of the object compiler into phases.

5- possibility of inclusion of external subroutines written in any language, and also possibility of precompiling several functions, or macros of the object compiler.

6 - AKNOWLEDGMENTS:

The design of the COMCOM system was influenced by the COMASS (compiler assembly system) written by R. Zarnke of the University of Waterloo, Canada. His work was super vised by Prof. D. D. Cowan. Many constraints of the original (COMASS) system were dropped, and several new features were included.

7 - BIBLIOGRAPHY:

1- R. Zarnke - "A Compiler and Software writing System", Internal report of University of Waterloo, 1965.

2- C. J. P. de Lucena - "A Software writing System", PUC, Series of Monographs in Computer Science Nº6/69.

3- S. E. R. de Carvalho - "Implementação do Sistema COMASS" - PUC, Master's thesis. Dec. 1969.

4- J. Slansky, M. Finkelstein, and E. C. Russel - "A Formalism for Program Translation" Journal of ACM, vol. 15 nº 2, pp-165-175, 1968.

5- J. Feldman, and D. Gries - "Translator writing Systems", Communications of ACM vol.11 nº 2, pp. 77-113, 1968.

6- S. V. Pollack and T. D. Sterling - "A Guide to PL/1", Holt, Rinehart and Winston, 1969.

7- J. E. Hopcroft and J. D. Ullman - "Formal languages and Their Relation to Automata", Addison Wesly, 1969.

8- S. Rosen ed. "Programming Systems & Languages". McGraw Hill , 1967.

9- D. E. Knuth - "The Art of Computer Programming"- vol.1,Addison Wesley, 1968.

10- F. R. A. Hopgood - "Compiling Techniques" , Mac Donald Computer Monographs, 1969.

11- T. B. Steel - "Formal Language Description Languages for Computer Programming" - North-Holland 1966.

12- H. Hagiwara, K. Watunabe: "Compiler Description Language COL" Information Processing in Japan, vol. 9, 1969.

13- M. V. Wilkes - "Computers then and Now" Journal of ACM. vol. 15 nº 1 pp. 1-7 , 1968.