

reASMgen

The Transformation of Assembler into COBOL85 or PL/1

1 Introduction

In the transformation of Assembler into a 3rd generation language (3GL) a gigantic range of problems has to be considered. At the one end of this range are the obvious 1:1 transformations, e.g. in PL/1

```
AP    FIELD, SUM
```

becomes

```
FIELD = FIELD + SUM
```

In the target language there is a 1:1 congruent construct. At the other end of this range are assembler constructs which even with the best of analysis cannot be automatically transformed into a 3GL language. Examples are operating system code and operating system macros.

One is faced with the question of how to determine which end of this range a given assembler program belongs to. Unfortunately this cannot be determined in a simple manner, e.g. by means of which assembler instructions have been used. The typical sequence in an operating system macro

L	15, XYZ	LOAD 15 WITH ENTRY ADR
BALR	14, 15	BRANCH TO ENTRY POINT

in general can **not** be interpreted, while it is expected that the following sequence from an application program will be interpreted as CALL UPB112.

```
L      15, =V(UPB112)
BALR   14, 15
```

This means: automatic transformation depends on the *style* of assembler programming. There is a style, varying from one company to another, in which commercial assembler programs are written and which in large part can be transformed. Many assembler programmers wrote programs in a style similar to COBOL but COBOL did not execute fast enough at the time of programming. A highly-developed assembler style (as e.g. that of SAP programs) cannot be transformed (or only with very style-specific rules). For example the use of multi-level pointers. Indeed, this could be transformed 1:1 into multi-level pointers (if the target language were e.g. C) but that is not what is expected. The above BALR could be transformed into a PL/1 CALL XYZ in which XYZ becomes a label variable but that is not what is expected.

It is expected that the program produced contain the language elements of a 3GL program to the extent possible.

The progress of 3rd generation languages over assembler consisted, among other things, in restricting the degrees of freedom of assembler and thereby bringing more semantic clarity into the program code.

E.g. in the instruction

```
BALR 14,15
```

register 15 is a variable (i.e. 1 degree of freedom). In order to determine where the flow of control branches the content of register 15 must be determined.

In contrast

```
CALL UPB112
```

no longer contains any degrees of freedom, the flow of control branches to procedure UPB112, the semantic statement of the program code is unambiguous.

The following describes how the tool **reASMgen** tries to bridge the semantic distance between assembler and 3rd generation languages.

2 Typing of Variables

In assembler the types of variable declarations often have little to do with the use of the variables in statements. In a 3GL, on the other hand, the use must agree with the declaration. The problem has the following variants:

- the type of field declaration does not agree with the type of field use
- the same field is used with different types at various places in the program
- a field isn't even declared, rather part of a larger field is referenced with offset + length
- there is no fixed-point number type in assembler, numeric fields just have an implicit decimal-point

reASMgen begins with the use of the variables and generates corresponding declarations. The ZAP in

```
STATMEG DS CL5
      ...
      ZAP REFELD2,STATMEG
```

shows that we are dealing with a packed field, thus **reASMgen** generates

```
DCL 01 STATMEG FIXED DEC(9);
```

In this case STATMEG is not used as anything other than packed; therefore the name STATMEG can be directly reinterpreted. But most of the time it is necessary to create a new interpretation as a redefinition:

```
REFE DS CL10
      ...
      MP REFE,=P'10'
      DP REFE,=P'6'
      ...
      UNPK REFE,REFELD20(5)
```

becomes in PL/1

```
DCL 01 REFE FIXED DEC(19) ;
DCL 01 filler2 BASED(ADDR(REFE)) ,
      5 REFE_1 FIXED DEC(17) ,
```

```

        5 REFE_9 FIXED DEC(1) ;
DCL 01 REFE_0 BASED(ADDR(REFE)) PIC '(10)9' ;
    ...
    REFE = REFE * 10;
    REFE_1 = REFE / 6;
    ...
    REFE_0 = REFE_1 / 20;

```

In particular the use of offset + length instead of a field declaration leads to extended sub-definitions:

```

        UNPK      IBUF+44(5),IBUF+22(2)

DCL 01 IBUF CHAR(80) ;
DCL 01 filler5 BASED(ADDR(IBUF)) ,
        5 filler6 PIC '(22)X' ,
        5 IBUF_22 FIXED DEC(3) ,
        5 filler7 PIC '(20)X' ,
        5 IBUF_44 PIC '(5)9' ;

```

An option can be used to specify whether, a NAME + OFFSET should be generated with the type **char** as *substring* (i.e. in COBOL: reference modification) or as *redefinition*. The option is specified in the profile REASM.PRO as follows:

```
options( substring_limit , «number» ).
```

If the same NAME + OFFSET formulation is used more often than the limit specified in the option, a redefinition is generated, otherwise a substring. From

```
MVC      PBUF+20(13),=C'SALES LIST  2'
```

the following is produced

```
MOVE 'SALES LIST  2' TO PBUF ( 21 : 13 ).
```

The **generated names** from a redefinition are produced by appending arbitrary combinations of offset and type to the original names until a new unique name is generated. Lower case is used deliberately, e.g. "filler72" or "ZW_dec", so that in revising the 3GL program the names can safely be replaced with an editor (there is no chance of confusion with possible names FILLER72 or ZW_DEC from the original assembler program).

reASMgen uses internally the following types:

- dec packed decimal
- bin binary
- num zoned decimal
- char character
- bits(2) single bits (here for example bit 2 within a byte)
- ptr pointer
- hex hexadecimal or unknown types

The type **bit**: as a rule the granularity with which **reASMgen** views memory is the byte. Bit operations are rare in commercial programming (in COBOL there aren't even any instructions for this). There are, however, **Switches** which are one byte in COBOL but in assembler typically one bit.

```
SW01      DS      XL1
```

```

...
XI      SW01,X'04'          toggle switch
...
OI      SW01,X'04'          set switch to one
...
NI      SW01,X'00'          set switch to zero

```

In the 3GL **reASMgen** declares one variable per bit referenced. The bit variable is redefined on the original variable so that memory use is not increased. Due to this the generated code will not function without manual intervention. It is necessary to check manually whether an increase in memory could cause problems (if e.g. SW01 is part of a database record). If not, the **REDEFINES** can be removed.

```

01 SW01 PIC X .
01 SW01-BIT-3 REDEFINES SW01 PIC X(1) .
...
IF SW01-BIT-3 = HIGH-VALUE
THEN
    MOVE LOW-VALUE TO SW01-BIT-3
ELSE
    MOVE HIGH-VALUE TO SW01-BIT-3
END-IF.
...
MOVE HIGH-VALUE TO SW01-BIT-3.
...
MOVE LOW-VALUE TO SW01.

```

It is not always possible to determine the data type from use in a single statement. For example from CVB

```
CVB      REG3,ZAELD
```

a type for REG3 and ZAELD can be determined, but not from

```
LA      4,3(3)
```

In this case the type is established through **data flow** investigations:

```
CVB      REG3,ZAELD
...
LA      4,3(3)

```

From the LA statement a reverse data flow search for REG3 is started. After a while it reaches the CVB statement and obtains the type of REG3 there. This is first entered in the right side of the LA statement and then, because it is an assignment, in the left side as well, i.e. in register 4.

The use of constants or EQUs is very helpful:

```
L      REG10,=A(BLANK)
```

The type of Reg10 can be determined as pointer in one step. The following construction, which at first glance seems similar,

```
ADR      DC      A(BLANK)
...
L      7,ADR

```

requires that the content of ADR be determined through a long data flow search. If the data flow search arrives at the beginning of the program, the type of ADR can be determined as pointer. Since data flow search for the purpose of type determination is also limited to a specific number of statements -- so that **reASMgen** doesn't use a lot of time for a possibly unsuccessful search -- it is possible that here the type of register 7 cannot be determined at all.

Data flow is an aid in **formatting for output** -- a very special data type. From

```
MASK      DC      XL12'40202020204B2021206B2020'
```

```

WRKSUMV  DC      PL5'0'
        ...
        MVC      PBUF+42(12),MASK
        ED       PBUF+42(12),WRKSUMV

```

the following is produced

```

01 filler180_0 REDEFINES PBUF.
   05 filler3 PIC X(42).
   05 PBUF-42 PIC BZZZZ.ZZ9,99.
01 WRKSUMV PIC S9(9) USAGE COMP-3 VALUE ZERO.
01 WRKSUMV-0 REDEFINES WRKSUMV PIC S9(7)V9(2) USAGE COMP-3.

```

```

...
MOVE WRKSUMV-0 TO PBUF-42.

```

in that beginning with ED the mask is sought backwards and then entered as data type in PBUF-42.

In assembler, numbers with **decimal positions** (e.g. WRKSUMV) only have the decimal point at an implicit location. The only indication of decimal positions is the mask in formatting for output. This is not enough to establish correct decimal placement for the entire program. If e.g. with WRKSUMV in general two decimal positions are reserved, it is still uncertain how to take this into consideration in an assignment `MP WRKSUMV, REFE`. Because the arithmetic of 3GL considers decimal definitions, an incorrect reservation of decimal positions would have the effect that the arithmetic executes erroneously. Thus also in the 3GL **reASMgen** retains an implicit decimal position and generates a decimal position redefinition for the MOVE appropriate for formatting for output.

3 The Formation of Data Structures

In assembler arbitrary redefinitions are allowed, sub-definitions are unknown. In PL/1 or COBOL on the other hand redefinitions are only allowed under strict criteria and sub-definitions provide an important element of style. **reASMgen** contains a few heuristics to achieve an organization with a natural appearance. From

```

QDSECT  DSECT
QREC    DS      0CL80
QSA      DS      CL2                      record type
QWARE    DS      CL11
QPRKILO  DS      PL4
QKILO    DS      PL4
          ORG     QWARE
QNAME    DS      CL10
QZIP     DS      CL4

```

for example, the following is produced in COBOL

```

01 QDSECT.
   05 QREC  PIC X(80) .
   05 filler9 REDEFINES QREC.
*       record type
         15 QSA PIC X(2) .
         15 filler72.
           20 QWARE PIC X(11) .
           20 QPRKILO PIC S9(7) USAGE COMP-3 .
           20 QKILO PIC S9(7) USAGE COMP-3 .
         15 filler69 REDEFINES filler72.

```

```

20 QNAME PIC X(10) .
20 QZIP PIC X(4) .

```

and in PL/1

```

DCL
01 QDSECT,
    05 QREC CHAR(80) ,
01 filler9 BASED(ADDR(QREC)) ,
    15 QSA CHAR(2) , /*          record type */
    15 filler72,
        20 QWARE CHAR(11) ,
        20 QPRKILO FIXED DEC(7) ,
        20 QKILO FIXED DEC(7) ,
01 filler69 BASED(ADDR(filler72)) ,
    20 QNAME CHAR(10) ,
    20 QZIP CHAR(4) ,

```

In general redefinitions in PL/1 are done with BASED(ADDR(xyz)). Since in assembler QREC could readily be used as a field, in the 3GL QREC cannot simply be used as a structure name; on the contrary, QREC must also be made available as a field in the 3GL. A generated name is redefined on it as a structure name. Since in assembler arbitrary redefinitions are allowed, the **reASMgen** heuristics cannot transform everything into the strict hierarchy necessary for a 3GL. In the LOG file there are references to such places. Sometimes in these cases a sub-definition can be achieved manually by raising the redefinition to level 01, while **reASMgen** unsuccessfully tried at level 15 or 20.

4 Explicitly addressed Variables

4.1 Tables

Explicitly addressed variables (also called "indirectly addressed variables") are memory locations addressed through registers, e.g.

```

ZAP    0(3,7),=P'7'

```

A transformation of assembler into a 3GL should use the language elements of the 3GL. Since assembler didn't yet have the constructs of indexed variables (or tables, arrays), the transformation should find and correspondingly interpret those assembler statements in which the programmer **intended** a table:

```

TAB7    DS      8PL3
...
LA      REG7,TAB7
...
ZAP     0(3,7),=P'50'

```

should become in PL/1

```

DCL 01 TAB7 (8) FIXED DEC(5) ;
...
REG7_TAB7_ix = 1;
...
TAB7( REG7_TAB7_ix ) = 50;

```

Unfortunately TAB7 is often not declared so clearly. Often one also finds e.g.

```

TAB7    DS      PL3
        DS      CL21

```

On the other hand not every indirectly addressed variable is a table. It can also be a case of character string processing or a pointer with some other, in principle arbitrary, use.

Thus it is necessary to use **heuristic rules** for indirect addressing that

- in those cases, in which a table was intended, map to a table
- in those cases, in which a table was **not** intended, do **not** map to a table.

This is a typical reengineering dilemma: a specific construct with specific semantics, that can be expressed explicitly in the target language, can be found in the source language only as programming style, mostly in different variants, a logical level deeper. One can try to recognize these patterns with heuristic rules, but must accept the fact that the rules will necessarily always be incomplete and the transformation is dependent on programming **style**.

According to our experience one of the few reliable indications for a table, other than the declaration

```
TAB      DS      8PL3
```

is use of an indirectly addressed variable in a loop. For example:

```
        LA      REG7,TAB
VM402   ZAP     0(3,REG7),=P'50'
        LA      REG7,3(REG7)
        BCT     REG11,VM402
```

or presented as an abstract pattern

```
        initializing REG with address
LABEL   using REG as pointer
        incrementing REG by increment amount
        GOTO    LABEL
```

A similar pattern is

```
        LA      REG7,TAB-3
VM402   LA      REG7,3(REG7)
        ZAP     0(3,REG7),=P'50'
        BCT     REG11,VM402
```

or presented abstractly

```
        Initializing REG with Address
LABEL   Incrementing REG by increment amount
        Using REG as pointer
        GOTO    LABEL
```

These patterns are best recognized through the **data flow** of the register. Here it is required that on the one hand the initializing of REG7 be unambiguously determined and on the other hand the incrementing by 3 in the loop be easily recognized. In the following (constructed) example the table is **not** recognized:

```
VM402   LA      7,1(7)
        ZAP     0(3,7),=P'7'
        LA      7,2(7)
        BCT     REG11,VM402
```

But with tables that do not have merely one field as table element but rather have a sub-structure e.g.

```
DCL      01 TABMKM_table (6) ,
          05 TABMKM_0 FIXED DEC(5) ,
          05 TABMKM_3 FIXED DEC(3) ,
          05 TABMKM_5 FIXED DEC(3) ;
```

it is not uncommon that in the loop the register is incremented at several places.

Tables with sub-structures are only rarely recognized by **reASMgen**, two- or multi-dimensional tables not at all. Further examples of non-transformable assembler code are pointer tables changed at run time, or multi-level use of pointers.

Once **reASMgen** has identified a table XYZ (e.g. through use in a loop), a subsequent heuristic starts and considers the surroundings of the declaration of XYZ, to see if further fields should be integrated into the table, e.g. the field

```

        DS      CL21
with
TAB7    DS      PL3
        DS      CL21

```

In this manner the size of the table is determined. Since the various combinations of assembler declarations limit what can be systematized in heuristics, in every case the programmer must manually check the size of tables (PL/1: dimension, COBOL: OCCURS).

Table recognition is the most complex program component in **reASMgen**. If you as programmer occasionally recognize a table that **reASMgen** didn't, keep in mind that

- humans are still superior to machines in pattern recognition
- in many other cases **reASMgen** quickly identifies tables with the aid of data flow search, while you as programmer -- very time-intensive and error-prone -- have to look for the sources of an indirectly addressed variable
- You can very easily identify for the tool **reASMgen** a table not recognized by inserting into the source code a redefinition such as

```
XYZ     DS      8PL3
```

or providing for the DS a reference number to the reference file containing a command for table interpretation.

4.2 The Processing of Character Strings

If an indirectly addressed variable is not to be interpreted as a table, it will be interpreted -- to the extent the data type permits -- as character string processing. Typical for character string processing is

- the use of a pointer that steps through the character string -- in assembler this is the register containing its address
- irregular incrementation of the pointer (in contrast to a table, with which the "pointer" is incremented regularly in a loop)
- access to the character string uses mostly pieces with a length of 1 byte

For example:

```

        LA      REG4,PBUF
        . . .
        MVC     10(3,R4),=3C'A'

```

becomes in COBOL

```

COMPUTE hptr = REG4-PBUF-ix + 10.
STRING 'AAA' DELIMITED BY SIZE INTO PBUF
                WITH POINTER hptr.

```

and in PL/1

```
SUBSTR(PBUF,REG4_PBUF_ix + 10,3) = (3)'A';
```

The semantics of tables and the semantics of character strings overlap: one-position tables can be interpreted as character strings; on the other hand character strings can be interpreted as one-

position tables. **reASMgen** takes advantage of this by simultaneously declaring for every character string a table -- thus for access with a length of 1, table access can be generated instead of the CPU-intensive PL/1 SUBSTR.

```
DCL 01 PBUF_value CHAR(133) ;
DCL 01 PBUF_table BASED(ADDR(PBUF_value)) ,
      03 PBUF (133) CHAR(1) ;
DCL 01 filler1 BASED(ADDR(PBUF_value)) ,
      10 ... sub-definitions.....
```

The above SUBSTR statement thus yields in PL/1

```
SUBSTR(PBUF_value,REG4_PBUF_ix + 10,3) = (3)'A';
```

4.3 DSECTs

DSECTs are essentially dealt with as indirectly addressed variables. Differences are:

- DSECTs have their own data structure hierarchy. If e.g. the DSECT CDSECT overlays the input record CUSTREC, this corresponds to a sub-organization of CUSTREC which **reASMgen** associates with CUSTREC through redefinition. A field such as CNAME can then be accessed in the 3GL through qualification (COBOL: OF, PL1: period).

```
CUSTREC DS      CL80
```

```
...
```

```
CDSECT  DSECT
```

```
CREC    DS      0CL80
```

```
CNUMBER DS      PL4
```

```
CNAME   DS      CL15
```

```
01 CUSTREC-value PIC X(80) .
```

```
01 CUSTREC-struct REDEFINES CUSTREC.
```

```
10 CREC PIC X(80) .
```

```
10 filler7 REDEFINES CREC.
```

```
20 CNUMBER PIC S9(7) USAGE COMP-3 .
```

```
20 CNAME PIC X(15) .
```

```
...
```

Use in COBOL: CNAME OF CUSTREC-struct

Use in PL/1: CUSTREC_struct.CNAME

- DSECTs are not used like indirectly addressed variables for character string processing.
- Positioning of a DSECT on a memory location that cannot be statically determined could be represented in COBOL85 with the new instruction SET ADDRESS ...TO... and in PL/1 with the positioning of a pointer. However, the address calculation that in assembler usually precedes positioning is not allowed in 3GLs.

4.4 Pointers

There are cases in which interpretation as a pointer would even be appropriate in the 3GL.

reASMgen can only do this in a very limited way, since there are no good discrimination criteria for the above interpretations as table/string -- from the view of assembler all of these cases are pointers. In particular **reASMgen** cannot mix the various pointer interpretations, e.g. to treat a register once as a pointer to be dereferenced and shortly thereafter as an index in string processing -- easily possible in assembler.

Passing **parameters** is a case requiring interpretation as a pointer. The parameters for the assembler program can be declared in the reference file. **reASMgen** includes the parameters in the program and interprets the dereferencing of the address list passed in REG1, e.g. in the declaration of two parameters of type address:

```
LM      3,4,0(1)
L        6,0(3)
L        7,0(4)
```

is interpreted in PL/1 as

```
DCL 01  REG3_ptr POINTER ;
DCL 01  REG4_ptr POINTER ;
DCL 01  REG6_ptr POINTER ;
DCL 01  REG7_ptr POINTER ;

.....
REG4_ptr = ADDR(parameter2) ;
REG3_ptr = ADDR(parameter1) ;
REG6_ptr = parameter1;
REG7_ptr = parameter2;
```

If -- as in the example -- the parameters passed are not data fields but addresses, as a rule **reASMgen** cannot deal with this correctly due to the multi-level dereferencing. In the example this works because REG6 and REG7 function as DSECT registers in the assembler program:

```
USING DS1,6
USING DS2,7
```

5 Flow of Control

5.1 The Uniqueness of Flow of Control

A more precise discussion of this problem can be found in the **reASM** handbook. **reASM** already contains all heuristics to determine the flow of control to the uniqueness required. For example with the construct

```
L      REG6,=A(PRINT)
...
BALR   REG14,REG6
```

through flow of control the content of register 6 is determined and -- if a unique content can be determined -- transformed into the PL/1 code

```
DCL 01 REG6_PRINT_ix POINTER ;
...
REG6_DRUCK_ix = ADDR(PRINT) ;
...
CALL PRINT;
```

The assignment of the address of PRINT is also generated, although in the 3GL this no longer makes sense. This is due to the fact that data flow only establishes a directed connection of the BALR statement to the L statement, but not the other way around. The BALR statement "knows" that the content of REG6 can only be the address of PRINT. The L statement does **not** "know" that the content of REG6 will only be used by the BALR statement and not later. Thus the L statement must be transformed. In PL/1 this is still part of the language, in COBOL no longer. In any case those situations in which **reASMgen** generates the data type POINTER must be checked manually. It can be a case of explicit addressing that could not be interpreted and must be corrected manually, or -- as here -- already correctly interpreted register usage, such that REG6_PRINT_ix can be deleted.

As you can see, many of the heuristics in **reASMgen** are based on **investigation of data flow**. Not only are these expensive in terms of time but -- especially in assembler -- their utility can be heavily impacted by unclear source code, in particular through flow of control anomalies (see **reASM** handbook). As a rule these should be removed from the source code before starting a **reASMgen** transformation.

5.2 Dynamic Code Modification

As already discussed in the **reASM** handbook, all places where code is modified are identified and -- to the extent possible -- re-interpreted as switches. While in **reASM** the emphasis is on identification and possible flow of control, in **reASMgen** generation as switches is important:

```
VT090      NOP      VERTRX
           OI        VT090+1,X'F0'
```

becomes in PL/1

```
VT090:
      IF switchVT090 = 1 THEN GOTO VERTRX;
      switchVT090 = 1;
```

and in COBOL

```
VT090.
      IF switchVT090 = 1 THEN GO TO VERTRX
      END-IF .
      MOVE 1 TO switchVT090.
```

5.3 Restructuring Flow of Control

reASMgen deliberately attempts *no automatic* restructuring of flow of control. Thus the initial result of transformation is 3GL code in which the correspondence between 3GL statement and assembler statement is easy to establish. This is important for manual control.

Once (perhaps after repeated corrections to the assembler source code) this 1:1 code is satisfactory, the restructuring module of **reASMgen** can be used to allow restructuring of selected code sequences. You will find a discussion of restructuring in the **reASM** handbook. **reASMgen** uses the same restructuring, except that instead of pseudocode PL/1 or COBOL85 is generated.

A much more important form of restructuring is the division of code into procedures. But this has little to do with the transformation of assembler into the 3GL. This form of restructuring bridges the distance between the "early", unstructured, 3GL programming style and the "late", mature, 3GL style. For this purpose a corresponding 3GL tool should be used (although one must add that the tools available for this are everything but satisfactory).

6 The Adaptability of the Transformation

6.1 Macros

Assembler macros are declared -- just as in **reASM** -- in the file MACRO.ARI. Now in addition, the 3GL code to be generated can be specified as needed.

It is not necessary to declare all macros but in comparison with **reASM** more macros must be declared. This is particularly true of macros that set a register. Since **reASMgen** investigates the complete data flow of the registers and uses this to obtain type information, the lack of information about the setting of a register has greater consequences than with **reASM**.

6.2 The Reference File

The reference file can influence the analysis/generation of individual statements. For this a reference number can be entered in the assembler source code or listing in the comment area. In the reference file an entry is made indicating how each reference number is to be interpreted. With the reference file you can for example

- introduce assembler statements to e.g. make the structure of an input buffer known through redefinition
- resolve ambiguities in the interpretation of an indirectly addressed variable
- declare tables
- deal with procedures with multiple entry points
- suppress the generation of a line or specify generation of text

A precise description can be found in the file REF.REF.

6.3 The generated Syntax

One of the design principles of **reASMgen** was that the differences between COBOL, PL/1, and other 3GL languages (the language C would have a special role here) have as little influence on the transformation process as possible. With few exceptions it was possible to defer the distinction between generating COBOL or PL/1 until the last step of the transformation process in which the internal metacode is turned into COBOL or PL/1 syntax. This step is in the file GEN.ARI and can be completely adapted, e.g. to generate other COBOL or PL/1 dialects, or instead of the COBOL85 STRING statements to generate company-specific subprogram calls for character string processing.

Experience with Prolog is required to adapt the file GEN.ARI.

Outside of the file GEN.ARI the difference between COBOL and PL/1 affects the following places (not adaptable):

- ordering data declarations, particularly redefinitions
- dealing with DSECTs
- if in an assignment the target variable would have to be a substring of a declared variable (common in PL/1, not in COBOL)
- if an indirectly addressed variable cannot be interpreted as table or string, in PL/1 a based pointer can be generated, not in COBOL