# technical memorandum Daresbury Laboratory

DL/CSE/TMD1

## SOME IDEAS ON WRITING EFFICIENT CODE IN PL/I

by

L. D. SMITH*
Daresbury Laboratory



*Present address:
Department of Computer Sciences, University of Edinburgh

January, 1977

Science Research Council
Daresbury Laboratory
Daresbury, Warrington WA4 4AD

## 1. INTRODUCTION

The opinions expressed in the following document are the result of nearly two years of attempting to write PL/I. Some of the views expressed will be controversial. In some cases I shall give evidence to support my views, and in others no evidence at all. You are free to accept or reject the lines I shall present - remember that they are personal views - but it is to be hoped that they provoke some thought.

The PL/I language reference manual (hereafter referred to as PLRM) has a complete chapter devoted to 'Efficient Programming' which gives many rules of thumb about how to write more efficient code. Unfortunately few general principles are given. Also, most of the advice is about how to make a particular construct execute most efficiently, rather than about how to avoid intrinsically inefficient constructs. I shall attempt to point to one or two areas that are intrinsically inefficient, and say why (of necessity) they are so.

## 2. ASSUMPTIONS

It will be assumed that the reader has some familiarity with PL/I or PL/I type languages (ALGOL, PASCAL, RTL2, etc). These languages are characterised by features such as block structure, implicit data conversion, many data types, powerful built-in and library functions, and being in some sense general purpose. Much of the following discussion will be equally applicable to other high level languages providing PL/I-like facilities. Some familiarity with the machine architecture and concepts of system-360 will also be assumed, but this will be at a fairly trivial level.



I shall take it as axiomatic that a major aim is to write well structured source code that reflects the structure of the problem under solution - a controversial axiom, no doubt. I shall not be concerned with any efficiency or inefficiency introduced by such structuring. I shall only say that it is my personal experience that my well structured programs are smaller, and compile into less code than my badly structured ones, and also that the potential for a compiler to optimise a program globally is increased if the program is well structured. In fact the stricter the structuring the more optimisation that can be performed.

## 3. WHY BOTHER TO WRITE EFFICIENT CODE IN A HIGH-LEVEL LANGUAGE?

Why should we trouble ourselves with writing good code in a high level language? Some schools of thought believe that as soon as we abandon assembler language we are condemned to bad code anyway. However, high level languages are quick and convenient to write in, and programs written in them are easier to maintain and debug, in general. This is not necessarily so - I have seen programs in PL/I that are almost incomprehensible, and assembler programs that read like plain English. Writing good code is also an end in itself - bad code offends us and good code is an art form.

It is my observation that, by using systematic programming methods, it is possible to effect a reduction in the size of object code generated from PL/I source code, of between 10% and 50%. The norm I use to measure this reduction from is a hypothetical non-expert user of PL/I, but one who is fairly fluent in some random subset of the language. It should be noted that PL/I is a large and rich language and that there are often several alternative ways to achieve a particular end (as stated in PLRM Chapter 18, page 254 etc), so it is usual for an individual to pick up

and use a fairly narrow subset of the language. An example might be a sandwich-course student who comes to Daresbury for a year, learns a subset of PL/I in a couple of months, and proceeds to write sizable programs - for example the GRASS system. This random approach to coding will in general lead to source code being larger and messier than is necessary, and object code being considerably larger than is necessary. Size of source code is, of course, an important consideration to those who have to (try to) maintain a system. I speculate that reductions of between 5% and 50% in source code size are in general possible with not too much effort. My experience re-writing the GRASS system was that notwithstanding removing trivially redundant and duplicated code, about 25% could be saved.

## 4. HOW DO INEFFICIENCIES APPEAR IN CODE GENERATED FROM HIGH-LEVEL SOURCE?

We will assume that some sort of optimising compiler is available to perform a reasonable amount of optimisation on the generated code, so we will NOT be concerned with topics such as global register allocation, common expression elimination, or moving of loop-invariant code. Rightly these are problems for the compiler alone and should be no concern of the programmer. We are going to look at those program constructs that cannot be reasonably optimised by an automatic process, but that could be spotted as potentially troublesome by a competent, though non-expert, programmer.

### 4.1 Poor Implementation

There is nothing the programmer can do about poor implementation of the compiler except grin and bear it. You could write to IBM but you are



not likely to get very far! An example of poor implementation might be the independent implementation of stream and record I/O in PL/I. Stream I/O could easily invoke record I/O (as in RTL2) to read and write records. Instead, separate modules are linked into the compiled program, thus increasing the load module size, and different modules are loaded from the runtime libraries to perform transmitter functions, thus increasing region requirements. There are also separate error messages, and separate chapters in the PL/I language reference manual. No doubt this is convenient for IBM and good for sales of main memory and manuals, but it leaves the newcomer to PL/I in a state of confusion.

Another example is the use of GETMAIN and FREEMAIN in PL/I (F) to get and free dynamic storage areas on entry to and exit from a PROCEDURE or BEGIN block. The overhead of this supervisor call, on all but the most massive PL/I procedures, is very large, and is partly responsible for the very poor performance of PL/I (F) compared with PL/I optimised.

If you are seriously worried by poor implementation then you should find another language to write in or code in assembler.

### 4.2 Providing Constructs that do not easily map onto the Target Machine

This is a major source of inefficiency in high level languages which are not machine oriented languages (MOL's) or system implementation languages (SIL's). An example is the manipulation of bit strings. PL/I defines bit strings of arbitrary length, beginning on an arbitrary bit of a byte. These are the unaligned bit strings, and (arguably unwisely!) this is the default for PL/I strings of all kinds (unaligned). This information is actually given in the PLRM (chapter 18, 'Efficient Programming'), but it is hidden away as a subsection or a minor section

(page 256 article 10 I believe). Also there is no reference to the colossal amount of code that will be executed by careless use of the UNALIGNED attribute.

Consider the problem of comparing two bit strings beginning at bits 2 and 5 of a byte respectively (see fig. 1). Remember that the 370 has no hardware support for bit addressing, so all access to arbitrary bits must be simulated using loading, shifting, and masking operations. Clearly even the assembler programmer does badly here and the solution is to avoid the problem if at all possible.

To this end PL/I provides a second type of bit string, namely ALIGNED bit strings. These commence on the first bit of a byte. Aligned bit strings whose lengths are a multiple of 8 may be handled as if they were characters, and the character manipulation instructions of the 370 may be used to handle them. For example, two equal length (< = 256 bytes) strings may be compared in a single CLC (Compare Logical Characters) instruction, whereas the same comparison for UNALIGNED data will take many hundred instructions in a library routine. Thus the programmer must be aware of the underlying hardware and not ask for impossibly inefficient operations if he wants the compiler to generate efficient code. It is of course arguable that high-level languages should protect the user from such 'sillies' but then either all semblance of machine independence disappears or arbitrary restrictions must be imposed. The best to hope for is a warning from the compiler, and it is an indictment of PL/I that no such warning is given, in this particular case. I suggest a message in the form of:

IELxxxI W 'BAD CODE GENERATED'



As a general rule:

(a) DO NOT use bit strings.

(b) DO use aligned bit strings.

(c) DO make the length of bit strings a multiple of 8.

(d) DO NOT use variable length bit strings.

Another example in this area is data-conversion. Some machines (e.g. ICL 1900 and 2900 series, MU5) have hardware assistance for fixed-float-fixed point data conversion. On the 370 between 6 and 7 machine instructions are required. However this is not necessarily disastrous as, on some 360 series CPU's, fixed and floating point operations are overlapped. Again the programmer must know his hardware in order to assess the cost of a particular operation.

Well, that seems to be enough about built-in and library functions. String manipulation, data-conversion, and I/O furnish many more examples of 'nice' high-level functions that have to be implemented in a clumsy way because of the hardware, so I shall say no more but refer the reader to PLRM (chapter 18, page 258 'Data Conversion') for many more examples. Let us look at another more fundamental difficulty occurring in all block structured languages. Let us look at how PL/I addresses its variables.

PL/I supports two major storage classes and several of lesser importance.

i) STATIC storage

This is used to contain program address constants, numeric constants, array, structure and string descriptors, etc. and all STATIC variables. Most of these, and all STATIC INITIAL variables, have initial values set up at compile time.

ii) AUTOMATIC storage

This is allocated at runtime from a stack (the Initial Storage Area or ISA). If the stack is not large enough PL/I attempts to obtain another segment from the user's region by issuing a GETMAIN. Although performance will never become as poor as that of PL/I (F), it is a good rule to ensure that the stack is as large as will be required. The runtime parameter ISA (nK) can be used to set the initial stack size, and the REPORT parameter can be used to find the size of stack needed. Otherwise the program may spend a lot of time thrashing through GETMAIN and FREEMAIN.

The use of AUTOMATIC storage allows space for compiler generated and user temporary variables to be reused by other procedures (see fig. 2). This can be a very significant saving of core requirements if the temporaries are large arrays. Note however that there are pitfalls. The use of AUTOMATIC INITIAL variables (implied by INITIAL) should be avoided as, in order to have an AUTOMATIC INITIAL variable, the compiler must store the value in STATIC and generate code to move the value into the automatic storage area at runtime (prologue code). All initialised variables should be declared as STATIC, unless it is genuinely required to re-initialise the variables upon each invocation of the procedure (or begin-block). It is arguable that the correct default for initialised variables should be STATIC, or at least that the compiler should issue a warning that it is about to generate bad code. However PL/I does neither of these things.

Now, each external procedure has its own STATIC area, addressed by a dedicated register (R8), so addressing STATIC variables is quite efficient. Note that internal procedures share the STATIC area of the outermost containing procedure.



Each invocation of a procedure also has its own automatic area, called a Dynamic Save Area, or DSA. This is used for the (guess what!) Register Save Area, for temporary variables, and for automatic variables. This is also addressed by a dedicated register (R13) so addressing of AUTOMATIC variables is also efficient.

Now, what if we wish to address the AUTOMATIC area of a calling procedure? So long as this is also a containing procedure, then the scope rules of PL/I allow this. Note that PL/I is not alone in this, and that the scope rules of almost all block structured languages allow access to variables declared in containing procedures (see fig. 3). Clearly a certain amount of work is required to calculate the address of a variable declared in a containing procedure. It is possible to construct pathalogical examples where access would be very inefficient, and certain architectures (e.g. 2900 series, MU5) have hardware support for stack addressing. 2900's support top-of-stack and segment before top-of-stack addressing which according to measurements made by Manchester University covers a very large percentage of all references. Manchester find that 70% of all references are to the top of stack, and a large proportion of the remainder are to global variables. The Burroughs higher-level-language machine has hardware addressing support for an incredible 20 stack levels, however.

It is recommended that certain simple rules are followed:

(a) make all global and initialised variables STATIC;

(b) make all local and temporary variables AUTOMATIC;

(c) avoid unnecessary back references to AUTOMATIC variables.

PLRM (chapter 18, page 255.9 f, q) gives the rules but does not explain why.

Access to STATIC EXTERNAL (FORTRAN COMMON) or CONTROLLED variables is less efficient still due to the extra addressing required: firstly the address of the external area must be loaded from STATIC (see fig. 4) or in the case of CONTROLLED variables, the address of the variable must be loaded from the Pseudo Register Vector (PRV) whose address is at a known offset in the Task Communications Area (TCA) addressed by dedicated register R12 (see fig. 5). PL/I will of course attempt to optimise the use of registers in this addressing, but if several EXTERNAL or CONTROLLED variables are to be accessed in a couple of statements, then PL/I runs out of registers to use and generates quite a lot of addressing code. PL/I also seems to miss quite obvious chances for optimisation as soon as things begin to get complicated, however I present no evidence for this claim.

BASED variables involve the same overhead as STATIC EXTERNAL (if the pointer is immediately accessible in STATIC or the DSA), but are often extremely convenient. In many cases PL/I handles chains of POINTERS almost as well as an assembler programmer might do, but of course fails to hold the most used values in registers as well as a programmer might do.

4.3 Tools for Program Checkout and Debugging

There are many features of PL/I that are useful for program checkout. We shall consider three of them.

i) DATA-directed I/O;

ii) ON-units;

iii) The statement number table.

Note also enabled check conditions, subscript range, string range, etc. These can easily be turned on and off without affecting program



logic so I have not included them as inefficient. Note however that library modules are always enabled for these conditions, so that choosing program constructs that are handled in-line will provide a worthwhile increase in speed. PLRM (chapter 18, page 257, article 11) mentions this point.

It is my personal view that once a program is debugged, all debugging aids should be removed from it. The main line program itself should detect all invalid data passed to it and initiate recover/abort procedures rather than relying on the runtime environment to trap silly data after it has caused a program crunch. PL/I's attempt to provide comprehensive runtime traps (ON-units) causes a significant loss of optimisation, compared with, say, FORTRAN-H.

DATA-directed I/O should be avoided in production programs as the I/O modules are bulky and the compiler must include a symbol table into the compiled program, which is also bulky. Likewise the statement number table which is a table of PL/I source statement numbers versus program code offsets which is held in the static area and used by the error monitor to associate a line number with the offset of a program interruption. If this table is deleted, between 5% and 25% of the total MODULE size can be saved, but the error monitor will then only give the offset of an interruption. However, the offset can very easily be associated with a statement number by specifying the compiler OFFSET option and having the statement number table printed but as part of the compilation listing. Use the NOGONUMBER (NGN) and NOGOSTMNT (NGS) options to inhibit inclusion of the statement number table into the compiled program. I have found that an average of 15% of program size may be saved in this way.

Finally avoid the use of ON-units for trapping hardware conditions such as underflow, overflow, zero divide, if these conditions are likely to occur frequently. The overhead of the ON-unit and its associated supervisor interruption is quite large (order of milliseconds) so it is better to check for very large, very small and zero values in your program, and make the patch-up code part of the mainstream program rather than part of an ON-unit. Other ON-conditions - e.g. ENDFILE, ENDPAGE - are software detected by the PL/I library modules, and do not cause a supervisor interruption, so are less inefficient. Often they are the only reasonable means to an end (e.g. use of ENDFILE) and are thus acceptable. See also the PLRM (chapter 18, 'Efficient Programming') for a detailed discussion of exactly how and why ON-units inhibit optimisation.

## 5. RECAP AND MISCELLANY

So far this discussion has been quite technical and possibly even confusing. Let us recap a bit. Many of the features of PL/I that make it so useful - block structure, powerful built-in and library functions, fewer restrictions on language constructs - are precisely those features that, at first sight, lead the compiler to behave like a cretin. The programmer should be aware that asking for features not supported directly by the hardware (e.g. stack addressing, bit addressing) are likely to be expensive. It should be noted that part of the reason why FORTRAN is so effectively optimised is that it is so limited and has so many arbitrary restrictions on it.

## 6. MISCELLANEOUS FACTORS INHIBITING OPTIMISATION

The availability of ON-units seriously inhibits global optimisation,



especially as the scope of an on-unit can be inherited by an external procedure at run time, thus preventing the compiler making any assumptions about whether or not a particular procedure will use on-units. ON-units inhibit optimisation of register usage by requiring that up to date values of variables are available in core for asynchronous inspection. Specifying the REORDER option on a procedure goes some way towards reducing the inhibition but global optimisation is still reduced. Statement labels also inhibit optimisation due to the possibility of a branch label by a called procedure (GO TO out of block). A label marks a discontinuity in the program graph, and no assumptions about register values may be made following the label, as it is not clear that the branch is from the same block even. Again it is arguable that GO TO out of block should be illegal, as it is at best an unstructured and messy construct, and has to be implemented by means of interpretive code. Note that labels inhibit optimisation even without GO TO out of block, as it is not clear where control may have come from when control is passed to a label. At least the compiler is unable to determine this. It is to be regretted that PL/I has no CASE statement, as the combination of CASE and IF-THEN-ELSE removes much of the need for LABEL variables, and would perhaps enable more effective optimisation to be performed.

To end with we will look at a few (mainly futile) entertaining examples of how the optimising compiler handles certain trivial programs. In each case the code generated is horrendous.

Figure 6(a) demonstrates the prologue code generated (24 bytes) to initialise an array of two AUTOMATIC LABEL variables. Figure 6(b) shows the language restriction that LABEL variables may not be STATIC INITIAL (because a LABEL is a pair of values, (address(label), address (DSA)),

and address (DSA) is not known at compile time). Other AUTOMATIC INITIAL variables incur similar overheads.

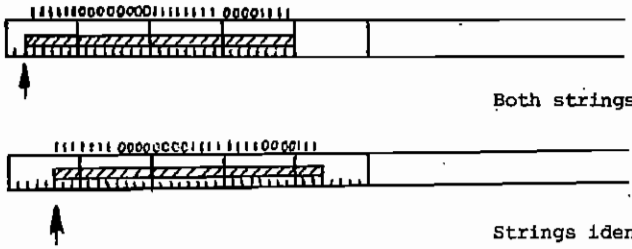
Figure 7(a) shows the (failed) attempt to ALLOCATE a variable length object in an AREA. Figure 7(b) shows how this may be done successfully by making the object 'self defining'. This provides no more actual security against FREEing the wrong length in the AREA, as LENGTH may be overwritten at will after allocation of the structure. Note that structure mapping is invoked to perform a trivial mapping, and that this includes 1.8k bytes of extra code into the program unnecessarily. This can be overcome by calling AREA management directly from a trivial assembly program.

Figure 8 shows a multiple entry-point, multiple exit-point procedure. At each textual exit (PL/I RETURN statement) an exit-code sequence is generated for each entry point. In this example 64 (8 x 8) exits are generated, together with code that ensures that only 8 of the 64 can ever be taken! In general, though not in this example, it would be possible for control to pass from any entry to any exit, depending on variable values supplied at run-time, so the compiler must generate code to cope with this, or allow the possibility of returning a value with the wrong attributes. Again, no warning message is given, and in this example 3470 bytes of code are generated.



FIGURE CAPTIONS

Byte 0, Bit 2



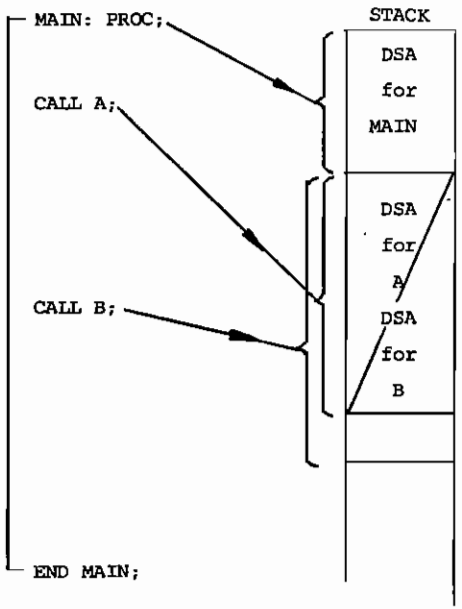
Both strings length 30 bits

Strings identical.

Byte 0, Bit 5

How do we compare these?

Fig. 1




SPACE RQD

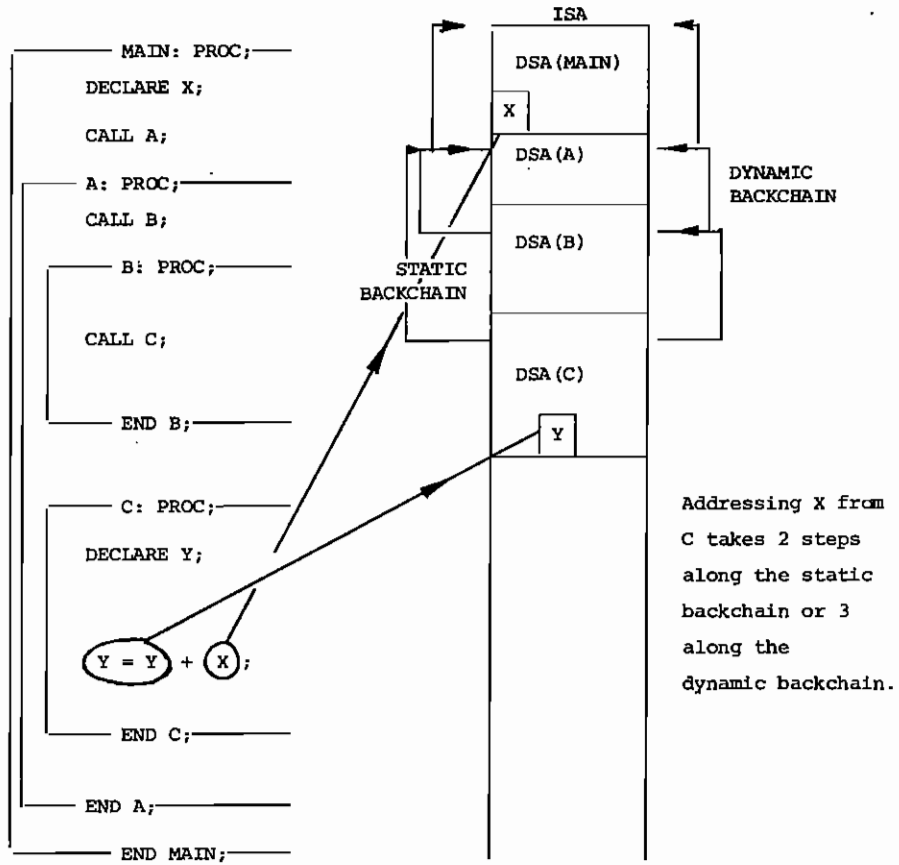
= DSA(MAIN)+
MAX(DSA(A),DSA(B))

<

DSA(MAIN) +
DSA(A) +
DSA(B)

(As in FORTRAN)

Fig. 2




Addressing X from C takes 2 steps along the static backchain or 3 along the dynamic backchain.

MAIN CALLS A CALLS B CALLS C;
C EXECUTES Y = Y + X;

Fig. 3

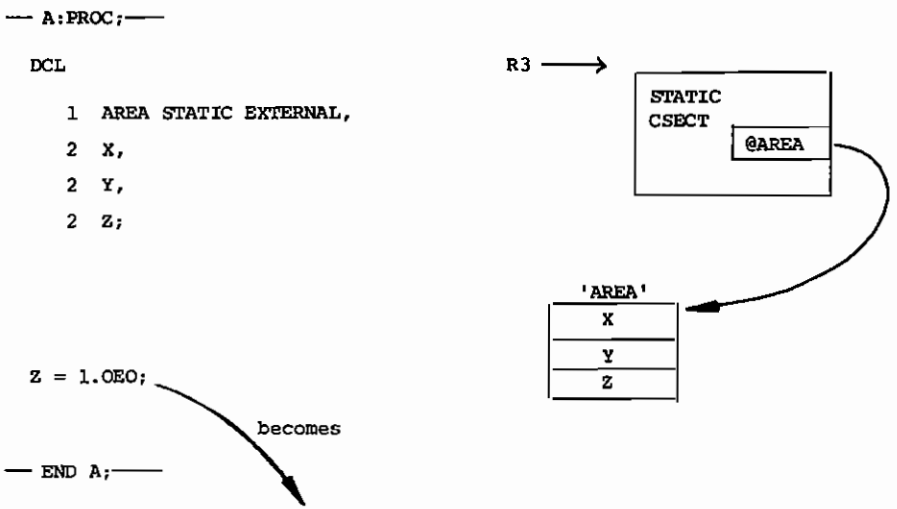
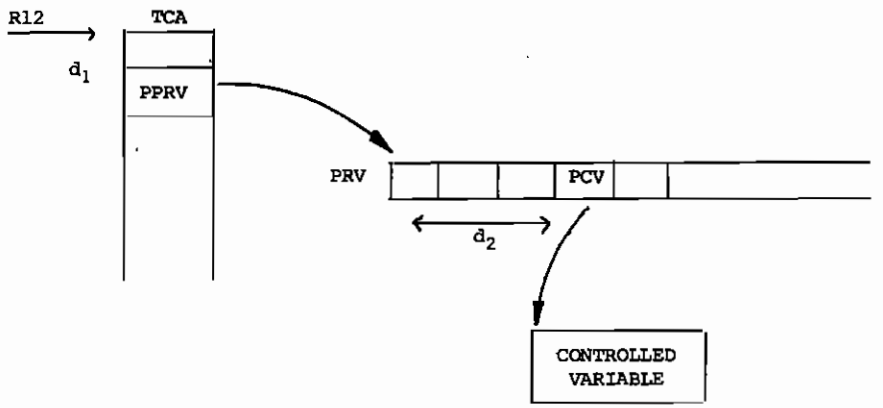



Fig. 4




$d_1$ offset of pointer to PRV known to compiler

$d_2$ offset of pointer to CONTROLLED VARIABLE known to compiler

PPRV filled in by PL/I initialisation

PCV filled in when variable is allocated.

CV = 1.0E0; TYPICALLY / BECOMES →

```
L   n,d (12)
L   m,d (n)
LE  x,d (3)
STE x,0(m)
```

Fig. 5

```
PL/I OPTIMIZING COMPILER    T:PROC(I)

                        SOURCE LISTING

NUMBER  LEV  NT

  10          0  T:PROC(I):                          'AUTOMATIC'
  20     1    0  DCL                                  VARIABLE
                     I FIXED BIN(15),
                     L(2) LABEL  INIT(L1,L2):
  60     1    0  GOTO L(I);
  70     1    0  L1:
                    STOP;
  90     1    0  L2:
                    RETURN;
 120     1    0  ENT T;

PROLOGUE BASE                                          24 bytes of prologue
INITIALIZATION CODE FOR L                              code to initialise 'L'
00006C   50  D0  D  OBC          ST    13,VO..L+12
00070    41  70  3  028          LA    7,40(0,3)
00074    50  70  D  0B8          ST    7,VO..L+8
00078    50  D0  D  0C4          ST    13,VO..L+20
0007C    41  70  3  030          LA    7,48(0,3)
00080    50  70  D  0C0          ST    7,VO..L+16
END OF INITIALIZATION CODE FOR L

00084    05  20                 BALR  2,0

PROCEDURE BASE

STATEMENT NUMBER          60

00086    58  70  D  0D0          L     7,208(0,13)
0008A    48  F0  7  000          LH    15,1
0008E    88  F0  0  003          SLA   15,3
00092    58  4F  D  080          L     4,VO..L(15)
00096    58  20  4  004          L     2,4(0,4)
0009A    58  40  4  000          L     4,0(0,4)
0009E    07  F4                  BR    4

STATEMENT NUMBER          70

STATEMENT LABEL      L1

000A0    41  10  3  024          LA    1,36(0,3)
000A4    58  F0  C  078          L     15,120(0,12)
000A8    05  8F                  BALR  14,15

STATEMENT NUMBER          90

*STATEMENT LABEL      L2

0000AA   18  0D                  LR    0,13
0000AC   58  D0  D  004          L     13,4(0,13)
000080   58  50  D  00C          L     14,12(0,13)
000084   98  2C  D  01C          LM    2,12,28(13)
000089   05  18                  BALR  1,14

*STATEMENT NUMBER         120

*FAO PROCEDURE
00008A   07  07                  NOPP  7

* END PROGRAM
```

Fig. 6(a)

```
PL/I OPTIMIZING COMPILER          T;PROC(I);

                        SOURCE LISTING

NUMBER  LEV  NT

  10          0  T:PROC(I):                          ATTEMPT TO GIVE STATIC
  20     1    0  DCL                                  VALUE FAILS.
                     I FIXED BIN(15),
                     L(2) LABEL STATIC INIT(L1,L2);
  60     1    0  GOTO L(I);
  70     1    0  L1:
                    STOP;
  90     1    0  L2:
                    RETURN;
 120     1    0  END T;

PROLOGUE BASE
00006C   05  20                 BALR  2,0          NOTE:  NO PROLOGUE CODE

PROCEDURE BASE

STATEMENT NUMBER          60

00006E   58  70  D  0C0          L     7,192(0,13)
000072   48  F0  7  000          LH    15,1
000076   8B  F0  0  003          SLA   15,3
00007A   58  4F  3  030          L     4,VO..L(15)
00007E   58  20  4  004          L     2,4(0,4)
000082   58  40  4  000          L     4,0(0,4)
000086   07  F4                  BR    4            'L' IS IN FACT UNINITIALISED

STATEMENT NUMBER          70

STATEMENT LABEL      L1
000088   41  10  3  024          LA    1,36(0,3)
00008C   58  F0  C  078          L     15,120(0,12)
000090   05  EF                  BALR  14,15

STATEMENT NUMBER          90

STATEMENT LABEL      L2
000092   18  0D                  LR    0,13
000094   58  D0  D  004          L     13,4(0,13)
000098   58  F0  D  00C          L     14,12(0,13)
00009C   98  2C  D  01C          LM    2,12,28(13)
0000A0   05  1E                  BALR  1,14

PL/I OPTIMIZING COMPILER          T:PROC(I):

COMPILER DIAGNOSTIC MESSAGES

ERROR  ID  L   NUMBER  MESSAGE DESCRIPTION

SEVERE AND ERROR DIAGNOSTIC MESSAGES          but compilation fails

IEL0580I E  20      INVALID INITIALIZATION FOR 'STATIC' LABEL 'L'.

COMPILER INFORMATORY MESSAGES

IEL0430I I  10      NO 'MAIN' OPTION ON PROCEDURE
IEL0541I I  10      'ORDER' MAY INHIBIT OPTIMIZATION

END OF COMPILER DIAGNOSTIC MESSAGES

COMPILE TIME  0.00 MINS      SPILL FILE:  0 RECORDS, SIZE  4051
```

Fig. 6(b)

```
PL/I OPTIMIZING COMPILER          ALLOCAT:PROC(P,L);

                    SOURCE LISTING

                                          Attempt to allocate a variable
NUMBER  LEV  NT                           length extent within an 'AREA'

  10         0   ALLOCAT:PROC(P,L);
  20     1   0   DCL
                   WSPACE STATIC EXTERNAL AREA(8192),
                   P POINTER,
                   L FIXED BIN(15),
                   SPACE CHAR(L) BASED(P);

  80     1   0   ALLOCATE SPACE IN(WSPACE) SET(P);
  90     1   0   RETURN;
 110     1   0   REPLACE:ENTRY(P,L);
 120     1   0   FREE SPACE IN(WSPACE);
 130     1   0   RETURN;
 150     1   0   END ALLOCAT;


PL/I OPTIMIZING COMPILER          ALLOCAT:PROC(P,L);

COMPILER DIAGNOSTIC MESSAGES

ERROR   ID  L   NUMBER   MESSAGE DESCRIPTION

SEVERE AND ERROR DIAGNOSTIC MESSAGES          COMPILATION ERROR

IEL453I     S     20     ADJUSTABLE EXTENT INVALID FOR BASED 'SPACE'

COMPILER INFORMATORY MESSAGES

IEL450I     I     10     NO 'MAIN' OPTION ON PROCEDURE

END OF COMPILER DIAGNOSTIC MESSAGES

COMPILE TIME     0.00 MINS        SPILL FILE:     0 RECORDS, SIZE  4051

COMPILATION ENDED BY 'NOCOMPILE' OPTION
```

Fig. 7(a)

```
PL/I OPTIMIZING COMPILER          ALLOCAT:PROC(P,L);

                    SOURCE LISTING

                                          Slightly altered program.
                                          Self-defining structure
NUMBER  LEV  NT                           allows variable length
                                          extent to be allocated.
  10         0   ALLOCAT:PROC(P,L);
  20     1   0   DCL
                   WSPACE STATIC EXTERNAL AREA(8192),
                   P POINTER
                   L FIXED BIN(15),
                   1 SPACE BASED(P),
                   2 LENGTH FIXED BIN(15),
                   2 FILLER CHAR(L REFER(LENGTH));
 100     1   0   ALLOCATE SPACE IN(WSPACE) SET(P);
 110     1   0   RETURN;
 130     1   0   REPLACE:ENTRY(P,L);
 140     1   0   FREE SPACE IN(WSPACE);
 150     1   0   RETURN;
 170     1   0   END ALLOCAT;

                                          'Length' is set to 'L' after
                                          the allocation is performed.

* PROCEDURE BASE
* STATEMENT NUMBER                100
0000BC                          CL.3    EQU   *
0000BC  D2  0B  D  0D8  3  03C          MVC   216(12,13),60(3)
0000C2  58  70  D  0C4                  L     7,196(0,13)
0000C6  48  60  7  000                  LH    6,L

0000CA  40  60  D  080                  STH   6,224(0,13)
0000CE  58  80  3  048                  L     8,72(0,3)
0000D2  50  8D  D  0F4                  ST    8,228(0,13)
0000D6  41  40  D  0F4                  LA    4,228(0,13)
0000DA  50  40  3  068                  ST    4,104(0,3)
0000DE  41  F0  0  008                  LA    15,216(0,13)
0000F2  50  F0  3  06C                  ST    15,408(0,3)
0000F6  41  10  3  068                  LA    1,104(0,3)        NOTE CALL TO STRUCTURE
0000FA  58  F0  3  018                  L     15,A..IBMBAMMB    MAPPING ROUTINE
0000FE  05  EF                          BALR  14,15
0000F0  58  40  D  0C0                  L     4,192(0,13)
0000F4  50  40  3  078                  ST    4,120(0,3)
0000F8  41  F0  D  0F4                  LA    15,228(0,13)
0000FC  50  F0  3  07C                  ST    15,124(0,3)
000100  41  10  3  074                  LA    1,116(0,3)
000104  58  F0  3  01C                  L     15,A..IBMBPAMA
000108  17  84                         LR    8,4
00010A  05  EF                          BALR  14,15       ---->   CALL TO 'AREA'
                                                                  MANAGEMENT
* INITIALIZATION CODE FOR LENGTH
00010C  58  80  4  000                  L     8,P
000110  48  60  7  000                  LH    6,L
000114  40  60  P  000                  STH   6,SPACE.LENGTH
* END OF INITIALIZATION CODE FOR LENGTH


* STATEMENT NUMBER                110
000118  18  0D                         LR    0,13
00011A  58  D0  D  004                  L     13,4(0,13)
00011E  58  F0  D  00C                  L     14,12(0,13)
000122  98  2C  D  01C                  LM    2,12,28(13)
```

Fig. 7(b)

```
PL/I OPTIMIZING COMPILER          ONE:PROC RETURNS(CHAR(1));

                    SOURCE LISTING

NUMBER  LEV  NT
  10          0   ONE:PROC RETURNS(CHAR(1));
  20     1    0   DCL
                    A1 CHAR(1) STATIC,
                    A2 CHAR(2) STATIC,
                    A3 CHAR(3) STATIC,
                    A4 CHAR(4) STATIC,
                    A5 CHAR(5) STATIC,
                    A6 CHAR(6) STATIC,
                    A7 CHAR(7) STATIC,
                    A8 CHAR(8) STATIC;
 110      1    0   RETURN(A1);
 120      1    0   TWO: ENTRY RETURNS(CHAR(2));
 130      1    0   RETURN(A2);
 140      1    0   THREE:ENTRY RETURNS(CHAR(3));
 150      1    0   RETURN(A3);
 160      1    0   FOUR:ENTRY RETURNS(CHAR(4));
 170      1    0   RETURN(A4);
 180      1    0   FIVE:ENTRY RETURNS(CHAR(5));
 190      1    0   RETURN(A5);
 200      1    0   SIX:ENTRY RETURNS(CHAR(6));
 210      1    0   RETURN(A6);
 220      1    0   SEVEN:ENTRY RETURNS(CHAR(7));
 230      1    0   RETURN(A7);
 240      1    0   EIGHT:ENTRY RETURNS(CHAR(8));
 250      1    0   RETURN(A8);
 260      1    0   END ONE;
```

Fig. 8