A. M. 100 PL/I Coding Conventions

This is the official A. M. 100 standpoint on PL/I. It is designed to help you write better and more scrutable PL/I code with a minimum of effort and extra time for debugging. Everyone reading this has undoubtedly heard of "GOTO-less" programming, but few have asked why this particular form of programming is used and encouraged. Well, as Aggie would say, "If there is anything we can't stand, it's some stupid program that might as well be given to the local rat for his nest." In other words, the flow of control in your program should be as "top-down" as your design. PL/I is a block-oriented language and you should make as much use of this facility as possible. In order to help you with structured programming, the following guidelines are, ummm, "encouraged".

First of all, always declare every variable in your program, even crummy DO loop counters and temporary work variables. Always declare every attribute of a given variable, i.e. FIXED BINARY (31,9) STATIC instead of just FIXED. Give your variables descriptive names, so that when one looks at the middle of the listing he won't have to try to remember what "X1" stands for. Declare each variable in a separate statement, and include a comment as to what it does. Along the lines of efficiency, most variables should be declared STATIC as opposed to AUTOMATIC (the default), especially if they are initialized. This will save both object program size and execution time. For the A. M. 100 program, you should use only FIXED BINARY (halfword or fullword) numbers (not float or decimal), and mostly non-varying character strings. If you need to use another data type, go ahead, but try to consider alternate methods. For example, a single bit flag is perfect for a DO WHILE condition, but declaring (and using) an array of bit strings generates code that would make you lose lunch.

Now then, GOTO's. DO NOT USE GOTO'S, except to exit an on-condition block. In many cases PL/I has no other means of doing this. All other places where a GOTO might come in handy can be replaced by either a DO WHILE loop, setting a bit value when you want to leave, or by a subroutine call. We hope to discuss top-down design and block structure from a more technical standpoint if anyone is interested. There is, believe it or not, an entire philosophy behind structured programming. If you follow it, you can save a great deal of time in design and debugging, which gives you time for other courses.
Take heed of PL/I's facilities and limitations regarding external procedures, parameters, and recognition of names. Subroutines should always be passed parameters explicitly, even if the subroutine is internal to the calling procedure. Try not to use the EXTERNAL attribute; it is also quite inefficient. Multiple declarations (variables which are declared in more than one external procedure), especially of structures, can be handled quite simply with PL/I's INCLUDE compile-time macro preprocessor facility (or with INCLUDE files from CMS).

ALWAYS end a particular DO group with an explicite END statement. This will avoid nesting errors. You need not label DO's and END's, but you should indent each group about five spaces when you begin it (start indenting after the DO;). Similarly, move back (to the left) five spaces for the END statement. This is not mandatory, as you may use six spaces if you desire.

Finally, try to keep implicit conversions to a minimum. PL/I conversions are not the best. For example, the statement "DO I=1 TO 32/2;" goes through the loop exactly zero times. Consequently, be careful using implicit conversions. Use the conversion functions if you're not sure what's going to happen in a given case.

Make use of the debugging techniques discussed in class, particularly hand-simulation and the use of "PUT DATA" statements and/or the CHECK option. Make careful use of ON-CONDITIONS. You will soon find that there is a tradeoff between space and time, and between programming ease and execution speed. Part of the purpose of giving a PL/I assignment is to help you become accustomed to these common coding problems. In addition, define some test conditions and write data to test them.

DOCUMENTATION and FORMATTING

As in assembly language, you should have a large block comment (in a box if you wish) at the top of your program. This comment should be verbose (complete) and should mention: the name and function of your program, its entry points (both internal and external), what parameters (name and meaning) are passed, what global assumptions are made, what special features and techniques are employed, and what error conditions may exist. It is intended that this comment be used to explain the function of your program to the next person who looks at it (i.e., a grader). If you come to a grader for help and your code is not commented, the grader will send you on your merry (or not so merry) way.

All DECLARE statements should be grouped together at the top of each procedure or BEGIN block, and well commented.
Each logical "block" of code within your program should also have a block comment. This comment should fully explain the function of the code it refers to, and should also explain how and why the function is implemented. A "block" is typically everything between "DO" and "END". There is usually no need for comments on individual source code lines unless the code is particularly complicated and it is possible (that is, there is room) to include neat, readable comments.

Column one of a source card can be used for carriage control, provided the option "SM=(2,72,1)" is used. This is a method of implementing "SPACE" and "EJECT" in PL/1. The codes are:

- (blank) skip to next line
- 0 insert one blank line in listing
- - insert two blank lines
- 1 skip to a new page

Finally, don't forget to indent DO groups and make your program aesthetically appealing. Anyone who looks at it later should see a neat, functional program that he wouldn't mind using himself.

Feel free to see a grader about any of the above. We will discuss it further if there are questions.

CJM