Published: 06/16/67

Identification

Implementation of Blocks in PL/I
D. B. Wagner

Purpose

The PL/I definition of a block is, "...a collection of statements that defines the program region--or scope--throughout which an identifier is used as a name. It also is used for control purposes" (IBM form C28-6751-3, p. 19, which should be seen for a detailed discussion of blocks from the point of view of the language). A block may be an external procedure, an internal procedure, or a begin block. The block structure of a program affects the organization of automatic storage (including calculation of adjustable extents and the initial attribute) and the actions of the on, revert, and go to statements.

According to the whim of the compiler, each generation of each block in a program may have its own stack frame, or the relation between blocks and stack frames may be more complicated. The present section discusses the various strategies which may be used to implement block-structuring, the mechanism for accessing automatic variables of containing blocks, and the mechanism for dealing with the "non-local go to".

External Procedures

Ignoring the fact that segments may be bound together (for which see The Binder, MSPM BX.14.01) external procedures correspond one-for-one with procedure segments. Every PL/I program segment is an external procedure with one or more external entries. The first label on the first procedure statement in the program segment is considered the "procedure name" and the "primary entry point"; This fact is noted in the Segment Symbol Table for the compiled segment and is otherwise ignored.

An external entry is of course called using the standard call-save-return sequences described in BD.7.02-.03. An entry beta in segment alpha is referenced using the notation

```
alpha$beta
```

and the simple name beta with no "$" refers to

```
beta$beta
```

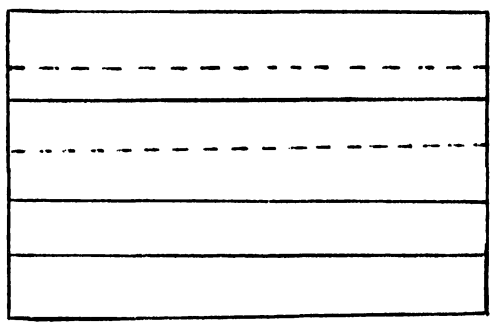
Internal Procedures

An internal procedure is called using the standard call-save-return sequences with the argument list modified by attaching to it the value of the stack pointer for the embracing block. This modification is described in BD.7.02. This implementation is mandatory for internal procedures that may be called from another segment, but an optimizing compiler may wish to use a special implementation for internal procedures which are not involved in intersegment communication. One possible special implementation is given later in the present Section. As an example consider the statements:

```
a:  begin;
b:  proc;
    ...
```

A call to b (no matter whence) includes in the argument list the stack pointer for the generation of storage for a. (Of course a must be active at the time of the call to b.) This stack pointer is used by b for accessing variables in embracing blocks.

This situation is complicated by the fact that an internal entry may be passed as an argument, and thus may be called from procedures which have no way of knowing the stack level of its embracing block or even whether it is internal or external. To solve this problem, whenever an entry is passed as an argument, the object passed is a six-word block (identical to a label) as follows:



ITS to entry

ITS giving stack pointer (for embracing block if internal; dummy if external)

2 spare words

Whenever any entry parameter is called, the caller includes the stack pointer value from the entry parameter in the same way as it would be passed in a call to an internal procedure. External procedures ignore this extra, while internal procedures use it to access variables in containing blocks without caring about its source.

Begin-blocks

It is reasonably clear why programmers use external and internal procedures, but it might be worthwhile to run through the reasons why a programmer might choose to use begin-blocks:

1. To keep identifiers used in different parts of a program from becoming confused.

2. To control when automatic adjustable strings and arrays have their extents calculated.

3. To control the execution of prologue and epilogue code for rarely entered sections of a program.

In addition the liberal use of begin-blocks in a program may provide valuable information to a smart compiler, since by definition they can be entered only at the top.

Begin-blocks: a simple strategy

To avoid complications in a simple compiler, a begin-block can be treated exactly like an internal procedure, so that the sequence

```
a:  begin;
    ...
    end a;
```

is compiled to be equivalent to

```
a:  call p;

p:  proc;
    ...
    end p;
```

The only problem which must be solved in a compiler using this strategy is that by PL/I rules a return statement inside a begin-block must terminate the closest containing procedure block and not just the begin-block.

A discussion of other strategies which might be used to reduce the amount of code the above strategy requires is discussed later in this Section.

Prologues, Epilogues, and the Non-Local Go To

Each block (external or internal) begins with a prologue and ends with an epilogue. The prologue performs various initialization tasks such as setting up the display, creating specifiers and (sometimes) dope for automatic variables, etc. It is not terribly important in this discussion.

The epilogue performs a number of tasks which must be done when a block is terminated. These include:

1) Reverting on-conditions
2) Freeing the storage occupied by automatic varying strings
3) Popping up the epilogue stack

The problem with epilogues is shown by the following series of statements:

```
a:   begin;
q:   ...
b:   begin;
     ...
c:   begin;
     ...
     go to q;
     end c;
     end b;
     end a;
```

The statement "go to q" is a non-local go to: when it is executed, the stack level must be brought down to the level of a; furthermore, the epilogues for both of the blocks c and b must be performed.

For the benefit of "abnormal returns" such as PL/I's non-local go to, Multics provides the unwinder. See BD.9.05 for details: it keeps a stack of terminating procedures corresponding to frames of the call stack. When a Multics procedure wishes to make a transfer of control which involves "popping" the call stack, it calls the unwinder, which executes the terminating procedure for every stack frame which must be violently terminated.

Thus in PL/I any go to statement which is not known to be local is compiled as:

```
call unwinder (l);
```

where l is a label variable (see BP.2.01) which contains the transfer point and the stack level associated with it.

For any block which needs an epilogue the epilogue is compiled as a standard internal procedure. Then in the prologue for this block code equivalent to the following is compiled:

```
call condition("cleanup", p);
```

It may come as a surprise to some readers that the condition primitive is used for this purpose. This is the way the unwinder works: see BD.9.05.

Begin-blocks= other strategies

The subject of precisely where the automatic storage for a begin-block is kept, and the relationship between blocks and stack frames, is a subject for discussion and possibly even for optimization decisions in the compiler. The extremes for consideration are:

1. As discussed above [and currently implemented in EPL, see BN.5.01], a begin-block may be considered to be simply a funny kind of internal procedure. Then there is a one-to-one correspondence between blocks and stack frames.

2. Begin-blocks can have almost nothing to do with stack frames: the automatic storage for the block can be kept in the stack frame for the most closely containing procedure block.

The first above is probably easier on the compiler, since it means a minimum of special cases to check for. It comes close to best-possible use of stack storage but requires quite a bit of extra code for communication with containing blocks (in EPL one instruction for each access to an automatic variable in a containing block, plus the code at block entry to create a new stack frame and set up the "display": see BN.6.03 and BN.6.04).

The second strategy above requires no special code for communicating with containing blocks. Its use of stack storage is normally less efficient than in the first case, but this inefficiency can be kept small if the compiler is reasonably clever about overlapping the storage used by parallel blocks.

The only problem with this second strategy concerns the non-local go to. The unwinder works only in terms of stack frames; if a begin-block does not have a stack frame of its own there is no way of telling the unwinder precisely under what circumstances its terminating procedure should be invoked. There appears to be a relatively straightforward way of handling this problem using a collection of switches telling which blocks using a given stack frame are currently active. At every point which might be the target of a non-local go to, a test can then be made to decide whether any descendants must be terminated before processing continues. Working out the details of how to do this may not turn out to be worth the trouble.

A simple way to avoid this problem is to make some begin-blocks work one way and some the other: Any begin-block which needs an epilogue has a stack frame of its own, and any other begin-block uses the same stack frame as its containing block. This is the strategy recommended for PL/I and PL/I-like compilers in Multics.

Internal Procedures: a special-case strategy

If an internal procedure is not involved in intersegment communication (i.e., is never passed as an argument to an external procedure), its calling sequence need not follow the system standard given in BD.7.02. The only requirement is that special calling sequences be documented and that the Segment Symbol Table output by the compiler be extended to include what calling sequence the procedure uses. Specification of the calling sequence suggested below is included in the standard PL/I Segment Symbol Table described in BD.1.02.

If all the calls to a given internal procedure come from its immediately containing block (this is probably true of 75 percent of all internal procedures) the procedure can be treated roughly like a begin-block without a stack frame. The call suggested is as follows: make up a standard argument list as in BD.7.02 but do not bother making the header which tells the number of arguments (let it be garbage). Then the call is simply

```
eapap   arglist
tsx7    proc
```

where proc is the procedure. This saves six instructions per call. Savings in the procedure itself depend upon strategies in other departments; in EPL the procedure would look like:

```
proc:   stpap    sp|arg
        stx7     sp|ret
        ...
        ldx7     sp|ret
        tra      0,7
```

this means the minimum savings would be: one instruction per reference to the containing block, two instructions in prologue and epilogue, and sixteen instructions in subroutines (execution saved only). The saving of execution time can be as much as twice that mentioned here depending on block level.