The PL/I *Checkout Compiler was designed to emphasize programmer productivity in developing programs, even at the expense of consuming extra machine resources. We explain the choices in the design of the compiler that resulted from this emphasis. The design is constrained by the requirement that a subroutine developed using this checkout compiler should be capable of executing in conjunction with code generated by a more conventional compiler. The execution environment that supports this operation is described.*

Design of a checkout compiler

by B. L. Marks

PL/I is a general-purpose computer programming language suitable for both scientific and commercial programming. It originated when the language was specified by a committee comprising representatives from IBM and the user organizations SHARE and GUIDE.[1] In both its initial design and subsequent development, the language aimed to provide a way of coding normal programming tasks quickly and efficiently.

The appearance of a programming language to its users depends not only on the language itself, but also vitally on the implementations of the language. The first implementation of PL/I, the PL/I F Compiler for Operating System/360 (OS/360), was introduced in 1966. Further facilities and performance have been added in four subsequent versions of the F compiler. In 1971, the F compiler was superseded by a pair of compilers, the PL/I Checkout Compiler and the PL/I Optimizing Compiler.[2,3] The Checkout Compiler, a program product that runs under OS/360, raises the programmer's productivity while he is creating a correct working program. If the working program is to be used repeatedly, the Optimizing Compiler will generate, from this same PL/I source program, object code to execute the program more quickly.

The use of two compilers operating on the same source program eliminates the compromises that an all-purpose compiler must make between assistance for the programmer and optimum use of the hardware. The idea of a pair of compilers is not new but has been carried further than ever before.[4]

In this paper, we discuss the choices in the design of the Checkout Compiler that resulted from emphasizing programmer productivity. We first discuss the aims, constraints, and decisions

Figure 1 Program with syntax errors

```
SOURCE LISTING
    STMT

       1  EX1:PROC;
       2     CLAL P1(X;
       3  P1:PROG(Y);
       4        PUT LIST('IN E1);
                PUT LST(Z)
          END EX1;
SYNTAX MESSAGES
IEM0379I E 2 'CALL' ASSUMED IN PLACE OF AN UNRECOGNISABLE KEYWORD IN
  'CLAL ? P1(X;'.
IEM0019I E 2 RIGHT PARENTHESIS MISSING IN 'CALL P1(X ? ;'. A RIGHT
  PARENTHESIS IS ASSUMED.
IEM0379I E 3 'PROC' ASSUMED IN PLACE OF AN UNRECOGNISABLE KEYWORD IN
  'P1:PROG ? (Y);'.
IEM0668I E 4 UNMATCHED QUOTATION MARKS. A QUOTE IS ASSUMED AT ' PUT
  LIST(   'IN E1  ? );                                    '.
IEM0379I E 4 'LIST' ASSUMED IN PLACE OF AN UNRECOGNISABLE KEYWORD IN
  'PUT LST ? (Z)END EX1;'.
IEM0021I E 4 SEMICOLON MISSING IN 'PUT LIST(Z) ? '. A SEMICOLON IS
  ASSUMED.
MESSAGES SUPPRESSED 5 M
END OF MESSAGES

T?
```

that influenced the design. Then we describe the mechanisms for implementation of the compiler. An outline of the implementation appears in the Appendix.

Aims

Programmer productivity is measured in terms of the time, effort, and inconvenience that the programmer undergoes while developing a correct working program to fulfill some need. Productivity is, of course, affected by factors not related to the compiler; for example, it is affected by the availability of the computer and whether or not a conversation system is used. The main influence of the compiler is in the treatment of programmer mistakes, that is, discrepancies between the behavior of the program and the behavior that the programmer intended.

Mistakes fall into three classes—static, dynamic, and logical. Static mistakes are those that can be identified by examination of the written program; the program as written is not a valid construction in the programming language. Figure 1 shows an example of static mistakes. Dynamic mistakes are defined as mistakes that become apparent as the program executes. An example is shown in Figure 2. Logical mistakes are cases where the program is a correct construction, executes without apparent irregularity, but does not fulfill the intention of the programmer. As an example, the following statement looks suspicious to a human but not to the compiler.

```
AVERAGE_PAY = MAN_COUNT / PAYROLL_TOTAL;
```

The problems for the compiler can be discerned from these examples. There are problems of detection; e.g., if an area of

Figure 2 Program with dynamic errors

```
SOURCE LISTING
    STMT

       1  EX2:PROC;
       2       DCL L(4) LABEL, I INIT(0);
       3       ON ERROR BEGIN;I=I+1;IF I<5 THEN GOTO L(I);END;

       7       DCL A BASED(PA);
       8       ALLOCATE A; FREE A; FREE A;

      11       DCL B BASED(PB) CHAR(4), C FIXED BIN(31);
      12  L(1):ALLOCATE B; PB=ADDR(C); FREE B;

      15       DCL AR AREA(100), CH200 BASED(PCH200) CHAR(200);
      16  L(2):ALLOCATE CH200 IN(AR);

      17       DCL CH5 CHAR(5), BT40 BIT(40);
      18  L(3):CH5='101A1'; BT40=CH5;

      20       DCL(FD,FXDO) FIXED DEC(15);
      21  L(4):FD=999999999999999; FXDO=FD*2;
      23  END EX2;
NO SYNTAX MESSAGES
MESSAGES SUPPRESSED 1 W
NO GLOBAL MESSAGES

IEN1254I 4X ONCODE=4053. BASED VARIABLE 'A' REFERENCES THROUGH
  POINTER 'PA' A FREED GENERATION OF STORAGE. 'ERROR' CONDITION
  RAISED.
    AT 10 IN EX2


IEN1253I 10X ONCODE=4052. ATTRIBUTES OF BASED VARIABLE 'B' DO NOT
  MATCH THE ATTRIBUTES OF THE GENERATION OF STORAGE REFERENCED
  THROUGH POINTER 'PB'. 'ERROR' CONDITION RAISED.
    AT 14 IN EX2


IEN1066I 14X ONCODE=360. 'AREA' CONDITION HAS BEEN RAISED DUE TO
  INSUFFICIENT CONTIGUOUS SPACE FOR BASED VARIABLE 'CH200' IN AREA
  'AR'. 'ERROR' CONDITION RAISED.
    AT 16 IN EX2


IEN1013I 19X ONCODE=615. ONSOURCE='101A1'. 'CONVERSION' CONDITION
  RAISED DUE TO A NON-BINARY CHARACTER BEING FOUND WHEN CONVERTING
  'CH5 VALUE='101A1'' TO BIT STRING 'BT40'. 'ERROR' CONDITION RAISED.
    AT 19 IN EX2


IEN1077I 24X ONCODE=310. 'FIXEDOVERFLOW' CONDITION RAISED DURING THE
  'MULTIPLY' OPERATION ON 'FD VALUE=999999999999999' AND 'CONSTANT
  '2', WITH 'FXDO' AS THE TARGET. 'ERROR' CONDITION RAISED.
    AT 22 IN EX2


IEN1184A 26X ONCODE=4. 'FINISH' CONDITION RAISED.
    AT 22 IN EX2

?
```

storage has been released to the operating system as available for reuse, how do we know when a subsequent reference is made to the data that was previously in that area of storage, as in Figure 2? There is a need for good description; e.g., in the first error message in Figure 2, the statement AT 10 IN EX2 identifies statement 10 as the failing statement and 4X indicates that four statements have been executed. And there are problems of correction; e.g., where to insert the "quote" mark in the example of Figure 1. Correction is an effort to guess the intention that has been hidden by the mistake, or to substitute some innocuous action so that processing can continue, before the programmer has corrected his mistake. Correction is necessarily *ad hoc;* as an example consider an invalid word that appears in a program in a position where the list of words that would be valid is short.

If the invalid word can be simply transformed into a valid one, such as by transposing two adjacent letters, then the valid word was probably intended. If the invalid word cannot be simply transformed into a valid one, then the mistake is probably undecipherable and an innocuous substitute is required.

To a limited extent, it may be possible to detect some examples of dynamic mistakes during compilation by predicting what the values of variables and the sequence of execution are going to be when the program executes. Since this prediction is difficult and not certain to find any error, the Checkout Compiler does the tests during execution instead.

Logical mistakes cannot be detected automatically. What can be done is to make the behavior of the program very obvious to the programmer. This can be done by facilities such as printing the value of a variable each time the value changes and giving the programmer a detailed history to compare with his expectations.

Constraints

The major design constraint was compatibility with the PL/I Optimizing Compiler. A program that has been checked out under the Checkout Compiler should be able to use all the language supported by the Optimizing Compiler and should run with the same results using the Optimizing Compiler. This compatibility was extended in two ways to ease the joint use of the compilers:

1. The programmer can combine the object code of procedures compiled by the two compilers. This allows a program to be progressively changed from one running with diagnostic checking into one running at maximum speed.

2. The same options are used in controlling the compilers, for example, to specify whether the compilation process is to produce a listing of the source program.

During the program checkout, there tend to be many compilations with short executions. This makes compilation speed important and execution speed less important. It was judged that an economically attractive checkout compiler would need to compile programs at least twice as fast as a more normal compiler and execute programs at generally no worse than one tenth the speed.

This compiler was to operate under OS/360. In its conversational variant it was to operate under the time-sharing option (TSO) of OS/360. Hence, constraints in main storage management and data set ownership were imposed by these systems.

Decisions

The major decision was to compile the PL/I source programs into an internal form and execute that form interpretively, rather than compile the PL/I program into instructions for System/360 hardware. This decision was based partly on the need for programs executing in conversational mode to be alterable during their execution but mostly on a consideration of diagnostics. Some of the actions connected with diagnostics were:

- Maintaining an identification of the PL/I statement being executed so that any error can be described accurately.

- Maintaining a count of the number of statements executed to trigger the action that has been asked for when the count reaches a limit. (When the limit is reached, a PL/I error condition is raised, and the program is allowed to continue for a limited number of statements. This allows for printing out details of the progress made by the program before it reached the limit. For programs that are erroneously executing an endless loop of instructions, this is more helpful than simply terminating the program when it has consumed a certain amount of central processing unit time. There is a similar limit on the number of lines of printed output.)

- Maintaining access to the original programmer-assigned name of a variable while operating on the value of the variable as encoded in main storage. This allows any error to be reported in the programmer's terms of reference.

- Checking that a variable has a value put in it by some operation before the variable is used as the source of a value input to some other operation.

It would be possible to compile System/360 code for this sort of activity. But because a sequence of System/360 instructions would need to be compiled for each occurrence of such pervasive activities as starting a new statement or accessing a value, the resulting code would be extremely large. In fact, one would probably have to compromise the extent of diagnosis to keep the size of the object program reasonable.

Organization as an interpreter ensures that code to perform diagnosis is required only once, in the relevant interpreter routing, irrespective of how many times the PL/I source programs require the function being checked. More opportunities for such checks are afforded by the full PL/I language than by simpler languages or scientific subsets of PL/I. Figure 3 depicts the operation under such an organization. The interpreter control moves the cursor

over the internal form of the program and invokes the appropriate interpreter routines that operate through the internal form and dictionary to process data.

internal form

There is great freedom of choice in deciding the internal form of the source program, the text that is being interpreted. We made the following choices:

- A three-address form in which temporary results are given explicit names, in preference to a Polish notation in which the temporaries are implicit, e.g.,

  $A = B + C \times D;$

- Three address form:

Multiply	*C*	*D*	Temporary
Add	*B*	Temporary	*A*

- Polish postfix late access form:

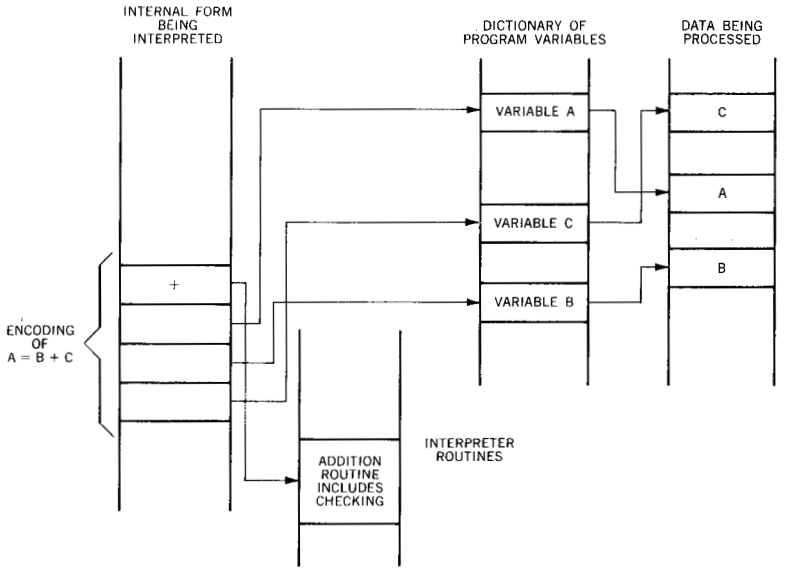
C
D
Multiply
B
Add
A
Assign

The more elegant Polish notation was rejected largely because of the execution-time cost of computing the attributes of the temporaries. For example, if A is a binary encoded number and B is a decimally encoded number, then the following formulas give the precision and scale of $A + B$. (See Reference 5 for the meaning of the parameters.)

$r = \text{MIN}\ (\text{CEIL}\ (PB \times 3.32) + 1, N1)$
$s = \text{CEIL}\ (\text{ABS}\ (QB) \times 3.32) \times \text{SIGN}(QB)$

$m = \text{MIN}\ (N, \text{MAX}\ (p\text{-}q, r\text{-}s) + \text{MAX}(q,s) + 1)$
$n = \text{MAX}(q,s)$

Figure 3 Organization as an interpreter



With the Polish notation, such attribute calculations would often be required during execution. With explicit temporaries, the calculations are made during compilation and recorded in the dictionary.

aggregate expansion

Another activity performed at compilation time to ease the interpretation load is aggregate expansion. In PL/I, most operations between aggregates, that is, arrays or structures, are defined in terms of an expansion into operations on the individual elements in the aggregate. By performing these expansions at compilation time, making the three-address code reference individual elements, we arranged that most interpreter routines did not need to handle aggregates. Notice that whereas this simplifies the interpreter, it does not speed up interpretation since it makes more text to be interpreted. It would be faster to have more routines that worked directly on aggregates.

Although temporaries are made explicit, considerable work is still left for the interpreter routines. For example, the ADD operator is polymorphic to the extent that while the arguments are known to be numeric and similarly encoded, there is not a unique ADD operator for each type of encoding. Hence, the ADD interpreter routine will need to discriminate, and the addition may ultimately be performed by a System/360 instruction for Add, Add Decimal, Add Halfword, Add Normalized Long, or Add Normalized Short within the interpreter routine. Also, alignment of the operands may be required to allow for scaling.

Diagnostic considerations also decided the split between checking at execution time and checking at compilation time. For example, if Z is a floating point variable and the programmer writes GOTO Z, this could be detected as an error at compilation time, or it could be left to the interpreter routine to find when executing the GOTO statement. Although it slows the compilation rate, there is a considerable advantage to making the test at compilation time. It ensures that the error is detected on the first processing of the program irrespective of how good or bad the test data used in executing the program is. If the checking is left until execution time, there is a risk that the part of the program in error will not be exercised. This principle is carried through to the extent of sometimes double checking, e.g., if a reference is made to an element of a matrix, using a subscript that is both constant and larger than the bounds of the matrix, the error will be detected at compilation time; and detected again in execution by the mechanism that checks the case where the subscript is not a constant.

subroutines

PL/I has a number of built-in functions, for example, the trigonometric functions SIN and COS, which are traditionally implemented by subroutines written by the compiler writers and supplied as the PL/I library. The subroutines needed for a particular program are bound with the compiled code for the program prior to its execution (Link Edit in OS/360 terms). To ensure compatibility with the Optimizing Compiler, the same subroutines are used by both compilers. This set of subroutines is called the Common Library. In the case of the Checkout Compiler, the subroutines are bound into the appropriate interpreter routine when the interpreter routine is originally constructed. Hence, the user does not require a Link Edit to include library subroutines, an economy in the use of the hardware.

PL/I allows the execution of a program to divide into tasks. These tasks then execute independently except where they are deliberately synchronized by the program. This makes it difficult to investigate errors, since the circumstances surrounding an error in one task may be changed by another task before there is time for the circumstances to be shown to the programmer. We decided on a level of automatic synchronization so that all tasks were suspended when one was in error. This is implemented by having an interpreter routine to dispatch the PL/I tasks instead of using OS/360 mechanisms. This routine is used to interpret the PL/I activities relating to tasking, such as the WAIT statement, and whenever used, it decides on the basis of priorities which PL/I task should continue execution. Since all the tasks in a program are known, it can test for the "deadly embrace" situation, where each task is waiting for the others to do something before it can proceed.

A program that attempts to use the value of a variable without previously giving a value to that variable is clearly in error. There are various ways of detecting this, none of them perfect. One method is to put a flag (i.e., a bit to record yes/no) in the dictionary entry for the variable, the flag indicating whether or not the variable has yet been assigned a value. The flag can then be checked when the variable is used. The shortcoming of this method is that data arrays, (for example a matrix) have only one dictionary entry. It follows that this method is unable to cope with situations where one element of a matrix has been given a value and another has not. The same shortcoming applies to any replicated data, such as the AUTOMATIC data of RECURSIVE procedures.

Another method of detecting uninitialized variables is to provide flags with the data so that there is one flag for each data item. This method has complications when parts of a variable can be accessed (e.g., a substring of a string of characters), but detection can be made perfect. Unfortunately the extra space occupied by the flags effectively alters the size of the data items. For the Checkout Compiler alone to do this would violate the compatability constraint. For the Optimizing Compiler to allocate space for flags it did not use would be contrary to the efficiency objectives of the Optimizing Compiler.

The method actually adopted is to put a bit pattern to indicate "uninitialized" within the space allocated to the data item. As an example of this, an unnormalized floating-point value is put in a floating-point data item to indicate that the item is uninitialized, since floating-point arithmetic in PL/I is done with normalized values. The method works perfectly when an "impossible value" can be found for the data type and even works reasonably well in some cases where every possible bit pattern is a possible real value for the data type. For instance, in the case of a character string, a value is used to denote "uninitialized," which will not normally be required by the program because it is not a character that normal printers can print. In the rare case of programs that do require the particular value to be available for a character, a compiler option can be selected that makes the character available for that use.

Mechanisms

swapping

The total size of the code required to interpret all of PL/I is about 250,000 bytes. Because a particular program will not use all the facilities of PL/I, the requirement for a particular program may be in the range of 120,000 bytes. And because the statements in any short section of the program tend to use less functions than the whole program, the transient requirement may be

60,000 bytes. The code must be in main storage to be executed, so economy in the use of main storage follows from adapting the set of interpreter routines, which is in main storage, to the progress of the program.

Similarly one can argue that only part of the internal form being interpreted need be in storage at any time, only part of the dictionaries, and only some of the data that the program is processing. There is a trend toward perfoming the swapping with hardware.[6,7] A software implementation has to be selective and has to consider the unit of swapping, the rate of change, and location freedom.

Because of the amount of it, interpreter code has to be swapped. The unit of swap is an interpreter routine, rather than a fixed number of bytes of code. This is because it is likely that if part of a routine for a particular function is going to be needed, then most of the code in the routine has a high chance of being needed, and also for convenience, since the interpreter routine will naturally exist on disk storage as a unit. Note that swap-out is unnecessary since the routine does not alter during execution, and so can be loaded from the disk copy whenever needed.

If the internal form of the program being interpreted is divided into equal segments of about 2000 bytes, these will be good units because the sequential nature of program execution makes it likely that if any part of the segment is exercised, then a significant part will be. The value of equal segments depends on the style in which people write programs. We have not done any experiments to judge whether other units, such as the code comprising a DO group, would be noticeably better. An individual item in a dictionary is not a good unit because it is only about 20 bytes in size, and an arbitrary segment of dictionary is not a good unit because there is insufficient correlation between reference to an item and reference to its neighbors. The complete set of items relating to a PL/I external procedure does form a natural unit because the symbols for that external procedure are irrelevant during the execution of another external procedure.

The rate at which the set of interpreter routines required to execute a program changes is an attribute of the particular program. However, most programs are organized so that relatively long continuous periods are spent in different functional subdivisions of the program.

Some items in storage depend for their successful use on the particular storage address at which they are located. This means that they cannot be simply swapped out of one location and into another. For example, an interpreter routine, which is System/360 machine code, may contain within itself the storage ad-

dress of one of the instructions in the routine. Each time such a routine is loaded into main storage this value has to be set up according to the location at which the routine is loaded. It is possible to write disciplined code that does not need relocation, but there is some loss in speed and flexibility which was not desirable for the Common Library.

A worst case occurs when one unit of swapping contains a storage address of something within another unit. This would occur if data was adaptively loaded and contained PL/I POINTER data.

On the basis of the considerations above, the Checkout Compiler uses swapping of interpreter routines, segments of the internal form being interpreted, and dictionaries for external procedures but does not swap the data being processed.

storage management

Our scheme of adaption required two facilities in main storage management not present in OS/360:

- The ability to extend an allocated area of storage contiguously. This is needed, for example, in building the dictionary because its size only becomes apparent as the declarations in the PL/I program are processed.

- A third state for an item in storage. The operating system regards storage as either free, i.e., having no contents and being available for use, or allocated and, hence, totally unavailable. Interpreter routines, for example, need a state that is stronger than free, (because it is desirable that the routine remain in storage until next use) but weaker than allocated (because there may not be enough storage for all routines to be allocated in storage).

These storage managment facilities were constructed from the more primitive operating system facilities.

mixtures of output

A program that is a mixture of Optimizing and Checkout Compiler output may refer to the same data from these different parts of the mixture. This can happen in several ways: the data may be a parameter passed from one procedure to another, or there may be an external variable known to both procedures, or a file may be written on by one procedure and read by the other. The internal representation of the data, both in storage and on a file, is chosen to be the same for both compilers. When parameters are passed from a procedure from the Optimizing Compiler to a Checkout one, what actually happens is that control passes to a section of Checkout Compiler output with register one containing the address of a list of the storage addresses of the parameters in accordance with OS/360 and OS/VS (virtual storage) conventions. The Checkout Compiler has set up the dictionary

items relating to the parameters to indicate that they should be reached by this indirect route. Hence, processing operates on the same data in the same representation.

When a Checkout procedure passes variables as parameters to one from the Optimizing Compiler, the dictionary items for the variables are not specially constructed. The interpreter routine that implements the call from one procedure to another uses the normal dictionary items to build the list of the storage addresses of the parameters. The code from the Optimizing Compiler then addresses the data via this list.

In the case of external variables, the Linkage Editor ensures that both procedures refer to the same data. Consider first the simpler case of two procedures from the Optimizing Compiler that refer to the same external variable. The output of the Optimizing Compiler consists of machine code and also control information. The control information uses the symbolic name of the variable and tells the OS/360 Linkage Editor of a location that is to be filled with the address of that external variable. Hence, each of the procedures can in execution pick up, from a location known to it, the address of the variable. In the case of a procedure from the Optimizing Compiler together with a Checkout procedure, the mechanism is essentially the same.

The checkout compilation does not produce machine code, but it does produce the control information. Thus no special action is required from the Linkage Editor to allow these procedures to share external variables.

Summary

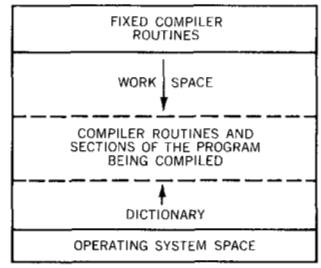
The Checkout Compiler was developed to work in harness with the Optimizing Compiler to provide an exceptional level of support for the PL/I language. The Checkout Compiler was designed particularly to increase the productivity of programmers in their development of programs. In meeting this objective, certain constraints had to be taken into consideration, chief of these being compatibility with the Optimizing Compiler. Some design choices were made for performance reasons.

CITED REFERENCES AND FOOTNOTE

1. G. Radin and H. P. Rogoway, "NPL: Highlights of a new programming language," *Communications of the ACM* **8**, No. 1, 9–17 (January 1965).
2. *OS PL/I Checkout Compiler: General Information Manual*, Form No. GC33-0003, IBM Corporation, Data Processing Division, White Plains, New York.
3. *OS PL/I Optimizing Compiler: General Information Manual*, Form No. GC33-0001, IBM Corporation, Data Processing Division, White Plains, New York.

4. F. C. Duncan, "Implementation of ALGOL 60 for the English Electric KDF9," *The Computer Journal* **5**, No. 2, 130–132 (1962).
5. *PL/I Language Specifications*, Form No. GY33-6003, IBM Corporation, Data Processing Division, White Plains, New York.
6. D. Sayre, "Is automatic folding of programs efficient enough to displace manual?" *Communications of the ACM* **12**, No. 12, 656–660 (December 1969).
7. Swapping as used here is not identical with page swapping used in the IBM System/370. The compiler was written initially to run under OS/360 rather than OS/VS.

Figure 4 Compile time storage layout



Appendix: Implementation of the Checkout Compiler

The compilation consists essentially of three passes over the program. The first pass reads in the source program, isolating each name, keyword, and operator. Most simple errors are detected by the syntactic analysis in this pass. The output of the first pass consists of two files, one for the DECLARE statements, one for the rest of the program.

The second pass over the DECLARE statements expands factored attributes, and the third pass enters the explicitly declared symbols in the dictionary. (If the LIKE attribute is used, there is an extra pass over the DECLARE statements to make the substitutions specified by the LIKE attribute.)

The second pass over the rest of the program relates the names used there to the corresponding dictionary items and adds dictionary items for names not explicitly declared. The third pass converts the program into the three address instructions that are to be interpreted.

It is theoretically possible to compile PL/I with just two passes over most of the program. However, the requirement that attributes should be available in a dictionary when the instructions are finally generated, and the fact that a declaration of a variable may come after the first appearance of the variable, make three passes a practical minimum.

The compile time storage layout, illustrated in Figure 4, allows for growth of the dictionary as declarations are processed, and for a fluctuating stack of workspace for the compiling process. The space between these two is competed for dynamically by the compiler's routines and the program being compiled. This dynamic behavior allows the compiler to make good use of whatever amount of storage is available.

The storage layout in execution is depicted in Figure 5.

Figure 5 Storage layout in execution