MAC TR-149

A PORTABLE COMPILER FOR THE LANGUAGE C

Alan Snyder

May 1975

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

A PORTABLE COMPILER FOR THE LANGUAGE C

by

Alan Snyder

ABSTRACT

This paper describes the implementation of a compiler for the programming language C. The compiler has been designed to be capable of producing assembly-language code for most register-oriented machines with only minor recoding. Most of the machine-dependent information used in code generation is contained in a set of tables which are constructed automatically from a machine description provided by the implementer. In the machine description, the implementer models the target machine by defining a machine-dependent abstract machine for which the code generator produces intermediate code. The abstract machine is abstract in that it is a C machine: its registers and memory are defined in terms of primitive C data types and its instructions perform basic C operations. The abstract machine is machine-dependent in that there is a close correspondence between the registers of the abstract machine and those of the target machine, and between the behavior of the abstract machine instructions and the corresponding target machine instructions or instruction sequences. The implementer defines the translation from an abstract machine program to a target machine program by providing in the machine description a set of simple macro definitions for the abstract machine instructions. In addition, macro definitions may be provided in the form of C routines where additional processing capability is needed.

This report is based on a thesis submitted to the Department of Electrical Engineering at the Massachusetts Institute of Technology on May 10, 1974 in partial fulfillment of the requirements for the Degrees of Bachelor of Science and Master of Science. Work reported herein was supported in part by the Bell Telephone Laboratories, Inc., the National Science Foundation Research Grant GJ-34671, IBM funds for research in Computer Science and by the Advanced Research Projects Agency of the Department of Defense under ARPA order no. 2095, ARPA Contract Number N00014-70-A-0362-0006 and ONR Task No. NR-049-189.

CONTENTS

1. Introduction

This paper describes the implementation of a compiler for the programming language C [1,2], an implementation language developed at Bell Laboratories and a descendant of the language BCPL [3]. The compiler has been designed to be capable of producing assembly-language code for most register-oriented machines with only minor recoding. Versions of the compiler exist for the Honeywell HIS-6000 and Digital Equipment Corporation PDP-10 computers.

C is a procedure-oriented language. It has four primitive data types (integers, characters, and single- and double-precision floating-point), four data type constructors (pointers, arrays, functions, and records), and a small but convenient set of control structures which encourage goto-less programming. An important characteristic of C is the minimal run-time support needed. Although C supports recursive procedures, C does not have built-in functions, I/O statements, block structure, string operations, dynamic arrays, dynamic storage allocation, or run-time type checking. The only run-time data structure is the stack of procedure activation records. Of course, to run any useful programs, an interface to the operating system is required, and a standard set of I/O routines has been defined in order to encourage portability. But the implementation of these routines is optional and separate from the task of implementing a C compiler which produces code for a given machine.

The compiler described in this paper was designed to be portable, that is, to be capable of generating code for many target machines with a minimum of recoding. When considering portability, three classes of machines can be defined:

1. Machines which can support C programs reasonably efficiently: This class of machines depends only upon one's interpretation of the term "reasonably efficiently." Clearly, all real machines can run C programs, limited only by some size constraint related to the availability of memory. However, the following capabilities are desirable: (1) the ability to access the current procedure activation record and the current argument list in a reentrant manner - this will require one or two base/index registers depending upon the calling sequence, (2) the ability to reference via a pointer variable - this will require another base/index register or an indirection facility, (3) character addressing, (4) integer arithmetic, and (5) floating-point arithmetic. Not all of the above capabilities need be present in the target machine; however, the more that are missing, the more interpretive becomes the execution of a C program. For example, the HIS-6000 is word-addressed; thus references to character variables are interpreted by a small run-time subroutine.

2. Machines for which the compiler can produce reasonably efficient code: This class of machines is clearly a subset of the first class; the size of the subset is again determined by one's definition of reasonable. The better the correspondence between the target machine and the machine model implicit in the compiler, the better will be the object code produced. On the other hand, if the correspondence is poor, the compiler may be able to produce only threaded code or instructions to be interpreted by software.

3. Machines which can support the compiler itself: Because the compiler is written in C, one may think that this class of machines is identical to the second class of machines; however, there are added restrictions which must be made in order to run the compiler on a given machine: the word size of the machine must be sufficient to hold all values used by the compiler; any implementation restriction on the size of procedures or data areas (as would be likely on the IBM S/360 because of addressing deficiencies) must not be such as to prohibit the proper execution of the compiler (this includes the ability of the compiler to compile itself). In addition, there are operating system and configuration restrictions: the memory size available to a program must be sufficient to hold the phases of the compiler; file space for the source of the compiler must be available and affordable; the I/O routines used by the compiler must be implemented. This class of machines is not a subset of the second class of machines since the compiler does not use all of the features of the language, notably floating-point.

This paper concentrates on the second class of machines, those for which the compiler can produce

reasonably efficient code, given the restrictions of the first class of machines, those which can support C programs reasonably efficiently. Thus, throughout this paper, the term "machine independence" will generally refer to the ability of a compiler to produce code for many machines.

1.1 Motivation

One of the serious problems in the field of software engineering is the difficulty of transferring programs to new machines. This is caused in large part by the proliferation of different programming languages and machines and the significant effort required to implement a compiler for any particular programming language and target machine. One approach to solving this problem is to restrict programming languages to a few standardized languages which are then implemented on all target machines of interest. A disadvantage of this approach is that it conflicts with the desirability of having many specialized languages for specialized problems. Another disadvantage is the fact that continual progress is being made in the development of programming languages so that by the time a language is standardized and widely available, it is already "obsolete." It is also difficult to achieve compatibility among the various implementations of a standardized language. Even if the standard language is well defined, it is difficult for compiler writers to restrain themselves from extending it and for users to restrain themselves from using the language extensions. A similar approach to the problem of program transferability is to restrict the number of target machines for which compilers must be written by requiring that each new machine be compatible with a widely-used existing machine. The stifling of progress in computer architecture which would result from this requirement is as undesirable as the stifling of progress in programming languages which would result from adoption of the previous approach. In addition, if the new machines are only upward compatible with the old machines, then problems may still remain with regard to transferring programs from new machines to old ones.

An alternative approach to those of language restriction and machine compatibility is to develop techniques that reduce the effort required to write compilers for various combinations of languages and machines. These techniques may be directed at two subproblems, that of reducing the effort involved in writing one particular compiler and that of reducing the effort involved in writing a family of related compilers. The development of such techniques could have benefits in addition to improving program transferability, such as making it easier to implement a new language or making languages more widely available.

An early effort in this direction was an attempt to devise a universal computer-oriented language UNCOL [4], which is both language-independent and machine-independent, to which all programming languages could be translated and which itself could be translated with acceptable efficiency into any machine language. The idea was that one need write only one UNCOL-to-machine language translator for each target machine and one source language-to-UNCOL translator for each source language, rather than having to write one compiler for each source language-machine language combination. In addition, if UNCOL were well defined, then the various implementations of UNCOL could be made compatible, thereby insuring the compatibility of the source language implementations. Unfortunately, the concept of a universal language has not led to a practical solution of the problem; the characteristics of source and machine language independence are incompatible with the need for acceptably efficient translation from UNCOL to machine language.

More practical techniques for reducing the effort involved in writing compilers result if one considers techniques with more limited goals than those of the UNCOL project. One approach is to develop techniques which reduce the effort involved in writing one particular compiler for some language-machine combination. Examples of such techniques are parser generators and syntax-directed symbol processors [5]. Another approach is to develop techniques for writing families of compilers for many source languages and one target machine. An example of such a technique is a compiler writing system with code generation primitives, such as FSL [6]. The third approach, and the one which is taken in this work, is that of the portable compiler, a compiler for a particular source language which can produce code for many target machines. It should be noted that techniques such as parser generators, which can aid in the implementation of a single compiler, can be equally useful in the implementation of more general systems such as compiler writing systems and portable compilers.

1.2 Background

A compiler can be considered to consist of two logical phases, analysis and generation. The analysis phase performs lexical and syntactic analysis of the source program, producing as output some convenient internal representation of the program, along with a set of tables containing lexical information and other information derived from the declarative statements of the program. The generation phase then transforms the internal representation into an object language program, using the information contained in the tables produced by the analysis phase. One can confine the machine (object language) dependencies of a compiler to the generation phase by a suitable choice of internal representation, i.e. one which is machine-independent. On the other hand, it is not practical to also confine the source language dependencies of a compiler to the analysis phase since this would make the internal representation a universal language. Thus the generation phase of a compiler is both source-language-dependent and machine-dependent.

Most portable compilers require that the generation phase be completely rewritten for each target machine [7,8]. This effort may represent only about one-fifth of the effort needed to rewrite the entire compiler [8]. In the case of the BCPL compiler [9], for example, moving the compiler may require only three to four weeks under ideal conditions (but otherwise may require up to five months). However, it would be desirable if the amount of recoding necessary to generate code for a new machine could be reduced.

One approach is that advocated by Poole and Waite for writing portable programs [10,11]. They advocate that before writing a program to solve a particular problem, one define an abstract machine for which the program is then written. With this approach, in order to move the program to a new machine, one need only implement the abstract machine on the target machine, typically via a macro processor. The desired qualities of the abstract machine are that it contain operations and data objects convenient for expressing the problem solution, that it be sufficiently close to the target machines of interest so that acceptable code can easily be generated, and that the tools for implementing the abstract machine be easily obtainable on the target machines.

This technique can be applied to portable compilers by considering the problem to be the implementation of an arbitrary source language program. The operations and data objects convenient for expressing the problem solution are then those which are basic to the source language. With this technique, a compiler would be broken into two parts: a machine-independent translator from the source language to the abstract machine language and a machine-dependent translator from the abstract machine language to the target machine language. The translator from the abstract machine language to the target machine language should be smaller and simpler than the conventional generation phase would be; typically, it consists of a set of macro definitions which map each abstract machine instruction into the corresponding target machine instruction or instruction sequence. Moving the compiler to a new machine simply requires rewriting the macro definitions.

The major difficulty with the abstract machine approach to portable software is in determining the appropriate abstract machine. If the abstract machine is of a high level (i.e., very problem-oriented), then the program will be very portable but the implementation of the abstract machine will be difficult. On the other hand, if the abstract machine is of a low level (i.e., more machine-oriented), then, unless it corresponds closely to the target machine, either the code produced will be inefficient or the implementation will be complicated by optimization code.

The solution to this difficulty proposed by Poole and Waite is to define a hierarchy of abstract machines, ranging from a high-level problem-oriented abstract machine to a low-level, machine-oriented, and easy-to-implement abstract machine. In this solution, the higher-level abstract machines are implemented in terms of the lower-level abstract machines, and only the lowest-level abstract machine need be implemented on a target machine in order to transfer the program; once it is transferred, higher-level abstract machines may be implemented directly in terms of the target machine in order to improve efficiency. While this technique may be useful for transferring particular programs, it is unlikely that it

will be acceptable in practical terms as a compilation technique because of the need for additional translation steps. An experiment by Brown [12] indicates that one may implement and then optimize a low-level abstract machine in about the same time as it takes to implement a higher-level abstract machine and that the resulting implementations are similarly efficient. Thus an alternative solution is to use a low-level abstract machine, but allow the implementer to optimize as desired; this solution is more likely to be acceptable as a compilation technique. A third solution will be advocated in this paper.

The technique of rewriting the generation phase requires that a non-trivial translator from the internal representation to the target machine language be written for each new target machine. Similarly, the abstract machine approach requires that a translator from the abstract machine language to the target machine language be written for each new target machine; if reasonably efficient code is desired and the abstract machine does not correspond very closely to the target machine, then this translator will also be non-trivial.

A more desirable goal for a portable compiler is that it have a generation phase which can be modified to produce code for a new target machine by a process which is largely automatic. Implicit in this goal is the requirement that the modification process obtain its knowledge about a target machine from a (non-procedural) description of the machine. An early effort in this direction was the SLANG system [13], which attacked the problem of describing a machine-dependent process (code generation) in a machine-independent way. In the SLANG system, source language constructs are translated into a set of basic operations called EMILs; the EMILs are translated into absolute machine code using macro definitions and instruction format definitions. The approach is similar to the abstract machine approach in that the EMILs can be considered to be the instructions of an abstract machine; the difference is that the code generation algorithm uses information contained in a machine description in order to tailor the EMIL program to the target machine. The EMILs differ from the instructions of a Poole and Waite abstract machine in that they are machine-oriented, rather than problem (source-language) oriented. In addition, the code generator does not seem to know about registers other than index registers, which implies that one will not be able to achieve the desired close correspondence between the abstract machine and most register-oriented machines. Nevertheless, the method of describing the instructions of a machine by providing simple instruction sequences which interpret the abstract machine instructions seems to be a good compromise between the desire to minimize coding and the difficulty of mathematically defining a machine and utilizing such a definition in generating code.

More recently, Miller [14] has explored the problem of constructing a code generator from a machine description. Miller proposes that a generation phase be constructed in two steps. In the first step, the language designer specifies the language-dependent part of the generation phase by writing a set of procedural machine-independent macro definitions for the operations of the internal representation produced by the analysis phase. These macro definitions define the operations of the internal representation, such as addition, in terms of machine-independent (i.e., language-oriented) primitives, such as integer addition, which are created by the language designer. In the second step, the implementer provides a description of the target machine which is used by an automatic code generation system named DMACS (Descriptive Macro System) in order to fill out the macro definitions of the first step and thereby produce a code generator for the target machine. As was the case with the SLANG system, the DMACS machine description defines the primitive operations by giving target machine code sequences which interpret them. In addition, however, the permitted locations of the operands (in terms of their being in memory or in particular registers) are specified as are the corresponding result locations. Thus the primitives can be made to correspond very closely to the instructions of the target machine so that the code sequences in the machine description are simpler and the resulting object code is more efficient.

Both the SLANG system and DMACS are intended to be general in that they are not designed for a specific source language. However, true generality is difficult to obtain and the systems do reflect preconceived notions about source languages. It is believed that, since there are much more significant variations among languages than among machines, a practical implementation of a compiler for any interesting language requires that the system be designed specifically for that language. This idea was recognized to some extent in DMACS where the primitives are created by the language designer as

convenient for expressing the operations of the source language. On the other hand, DMACS contains no notion of storage classes (different mechanisms for accessing variables of the same data type) which are needed for C; the implementation of storage classes is machine-dependent and thus must be defined in the machine description. In this paper, techniques similar to those used in the SLANG system and in DMACS are used in the implementation of a portable C compiler.

1.3 Method

The goal of this research is to design a generation phase for a C compiler which can be modified to produce code for many machines by a process which is largely automatic. Some insight into this problem can be gained by examining the corresponding, but better understood problem of the automatic construction of an analysis phase. One common approach is the use of a parser generator [15]. A parser generator is a program which accepts as input a grammar for a source language and produces as output a set of tables which are used by a language-independent parsing algorithm. The parsing algorithm is supplemented by a set of action routines which are provided by the implementer; these action routines are called by the parsing algorithm at appropriate points to produce the output of the analysis phase. The important characteristics of this process are as follows:

1. The analysis phase is divided into two parts, a language-independent part (the parsing algorithm) and a language-dependent part (the parsing tables and the action routines).

2. The language-dependent tables are constructed automatically from a finite description of the language (the grammar).

3. The analysis phase is "filled-in" by the implementer by providing information in a procedural form (the action routines).

4. The choice of a specific parsing algorithm determines the class of languages which can be handled by the analysis phase.

The process of constructing an analysis phase can be made more automatic through the use of a compiler writing system. In a compiler writing system, the action routines are in a sense built-in; the implementer invokes these action routines from a higher-level description of the translation. The use of such a system may involve much less effort than would be required to write a complete set of action routines. However, the important point here is that the use of built-in knowledge, as opposed to allowing the addition of arbitrary procedural knowledge, restricts the class of translations (and thus source languages) which can be handled by the automatically generated analysis phase.

For the compiler described in this paper, techniques analogous to those described in the preceding paragraph are used in the implementation of the generation phase. The generation phase is split into two parts, a machine-independent part and a machine-dependent part. The machine-independent part of the generation phase is a machine-independent code generation algorithm, corresponding to the language-independent parsing algorithm of the analysis phase. Just as the choice of a particular parsing algorithm limits the class of languages that the analysis phase can handle (the parsing algorithm is not completely language-independent), the choice of a particular code generation algorithm determines the class of machines for which the compiler can produce reasonable (non-interpretive) code. The machine-dependent part of the generation phase consists of a set of tables produced automatically by a stand-alone program GT (Generate Tables) from a machine description, which corresponds to the grammar in the construction of an analysis phase. The information contained in the machine description may be supplemented by a set of routines which correspond to the action routines of the analysis phase. However, the compiler described in this paper is closer to the compiler writing system approach in that implementer-supplied routines form only a minor part of the generation phase. The extent to which the implementer can easily and safely include such routines in the generation phase represents another factor determining the class of target machines handled.

A code generation algorithm, if it is to be machine-independent, requires a model of a machine with which to work. This model may express such notions as memory, registers, addressing, operations, and hardware data types. In the machine description, the implementer defines his target machine in terms of this model and also specifies the form of the object language. The class of machines for which the code generator can produce acceptable code directly corresponds to the generality of the machine model.

The machine model used by the C compiler is a C machine: a machine whose registers and memory are described in terms of the primitive C data types and whose operations are primitive C operations. The implementer models the target machine in terms of a C machine, producing an abstract machine. The abstract machine may be very similar to or very different from the target machine, depending upon how closely the target machine fits the machine model. The code generation algorithm, using its machine model, produces code for the abstract machine. The "assembly" language of the abstract machine is called the intermediate language; an intermediate language program, which is in the form of a series of macro calls, is translated into the target machine assembly language using a set of macro definitions, provided by the implementer in the machine description. Assembly language was chosen over machine language for the output of the compiler because it is far easier to describe and produce in a machine-independent manner than machine code or object modules.

The abstract C machine plays the same role in the C compiler as would a Poole and Waite abstract machine. The difference is that instead of there being one fixed abstract machine, there is a class of abstract machines, corresponding to the variability in the machine model. This variability allows the implementer to define a particular abstract machine which more closely resembles his target machine. The result is that the translation from the abstract machine language to the target machine language becomes simpler, and more efficient code is produced.

The process of modeling the target machine is described in chapter two. A detailed discussion of the code generation algorithm is presented in chapter three. Conclusions are presented in chapter four.

2. Modeling the Target Machine

The code generator's model of a machine is an abstract C machine, a machine whose instructions perform the primitive operations of the C language. The data types of the abstract machine are the primitive C data types (characters, integers, and single- and double-precision floating point), supplemented by one or more pointer classes which are distinguished by their ability to resolve addresses. The basic addressable unit of the abstract machine memory is the byte, which holds a single character value (characters are the smallest C data type). Values of the other abstract machine data types occupy an integral number of bytes, possibly aligned in larger units of memory. The abstract machine has a set of registers which may be used to hold the operands of the abstract machine instructions. Each abstract machine register is capable of holding values of some subset of the abstract machine data types. The instructions of the abstract machine are three-address instructions. Each address may specify an abstract machine register or a location in memory; the mechanisms for referencing a memory location correspond to the primitive addressing modes in C.

In the machine description, the implementer describes the target machine in terms of this machine model by defining a particular abstract machine for which the code generator produces intermediate code. The implementer specifies the sizes and alignments of the primitive C data types and defines pointer classes as convenient. The implementer defines the abstract machine registers, which generally correspond to those registers of the target machine which are to be used in the evaluation of expressions. The implementer also specifies the registers which may hold values of each of the abstract machine data types. In addition, the implementer may specify that any two abstract machine registers conflict in the target machine, meaning that only one may hold a value at any one time. The implementer defines the abstract machine instructions in terms of their operand/result locations and possible side-effects on other registers. In addition, the implementer provides a set of macro definitions which implement the abstract machine instructions on the target machine.

2.1 The Intermediate Language

The intermediate language is the assembly language of the abstract machine. Using the information contained in the tables constructed from the machine description, the code generator produces a translation of the source program in the intermediate language. An intermediate language program consists of a sequence of macro calls, each of which is expanded into one or more object language statements using the macro definitions provided in the machine description. There are two types of macros in the intermediate language: The first type are macros which represent the three-address abstract machine instructions. The second type are keyword macros which correspond to either assembly-language pseudo-operations or instructions implementing the primitive C control structures.

2.1.1 Abstract Machine Instructions

The abstract machine instructions are three-address instructions which perform the evaluation of C expressions. The operators of the abstract machine instructions are called abstract machine operators (AMOPs), the addresses are called references (REFs).

2.1.1.1 AMOPs

AMOPs are basic C operations which are qualified by the specific abstract machine data types of their operands. For example, in the HIS-6000 implementation there are four AMOPs corresponding to the C operator '+':

+i	integer addition
+d	double-precision floating-point addition
+p0	addition of an integer to a pointer to a byte-aligned object
+p1	addition of an integer to a pointer to a word-aligned object

In addition, there are AMOPs for data movement, data type conversion, and conditional jumps. AMOPs are represented in the compiler as an integer opcode with a value from 0 to 255. The various AMOPs are listed in Appendix II.

2.1.1.2 REFs

A REF is a C-oriented description of the location of an operand or the result of an abstract machine instruction. A REF may specify either a register of the abstract machine or a location in memory; the possible classes of memory references include C variables of various storage classes (automatic, static, external, parameter, temporary) as well as constants and indirect references. A REF is represented by a pair of integers called REF.BASE and REF.OFFSET; REF.BASE determines either a particular register or a particular class of memory references, REF.OFFSET determines the exact location given a specific memory reference class. The possible values of REF.BASE are listed below with their interpretations (actual integer values are shown for concreteness; the compiler itself uses manifest constants):

REF.BASE	Interpretation
n ≥ 0	- register *n* (register numbers are assigned to the registers of the abstract machine in a predictable manner by GT)
-1	- an automatic or temporary variable; OFFSET is the offset of the variable in the stack frame
-2	- an external variable, referenced by name; OFFSET is an internal identifier number
-3	- a static (internal) variable; OFFSET is an internal static variable number
-4	- a parameter; OFFSET is the offset of the variable or its address in the argument list
-5	- a label; OFFSET is an internal label number
-6	- an integer constant whose value is OFFSET
-7	- a floating-point constant; OFFSET is an internal constant number
-8	- a character string constant; OFFSET is an internal string number
n ≤ -9	- reference indirect through a pointer in register *n* (-n -9); OFFSET is the offset of the reference relative to the pointer

The specific values of REF.BASE need not be referred to in most macro definitions; the exception is the NAME macro, which converts a REF into a symbolic address.

The representation of a three-address instruction in the intermediate language is that of a macro call with five or seven integer arguments representing the AMOP and REFs for the result and the operands of the AMOP. (Each REF consists of two arguments, REF.BASE and REF.OFFSET; only two REFs are provided in the case of a unary AMOP.) The macro name used in the macro call is of a special form which specifies an entry in a table produced from the machine description by the GT program; this table entry refers to the representation of the corresponding macro definition from the machine description.

2.1.2 Keyword Macros

Keyword macros are those macro calls which, along with the three-address instructions, make up an intermediate language program. Unlike AMOP macros whose names are generated by GT, the names of the keyword macros are predefined, as are their functions. For example, keyword macros are used to define external variable names and internal labels; to specify initial values in storage, and to produce the function prologs and epilogs. The various keyword macros defined in the intermediate language are listed below along with a brief description of their functions; a more complete set of descriptions appears in Appendix III.

macro	function
HEAD	produce header statements, if needed
ENTRY	define an entry point
EXTRN	define an external reference
INT	define an integer constant
CHAR	define a character constant
FLOAT	define a floating-point constant
NFLOAT	define a negative floating-point constant
DOUBLE	define a double-precision float constant
NDOUBLE	define a negative double-precision constant
ADCONn	define a class "n" pointer constant
STRCON	define a pointer referencing a string constant
EQU	define a symbol
ZERO	define an area of storage initialized to zero
STATIC	define a static variable
STRING	define the string constants
ALIGN	force an alignment of the location counter
LN	define a line-number symbol
LABCON	define a label constant
LABDEF	define an internal label
IDN	translate an internal identifier number into the corresponding assembler symbol
END	produce an end statement, if needed
PROLOG	produce the prolog code of a C function
EPILOG	produce the epilog code of a C function
CALL	produce a function call
RETURN	produce code for a return statement
GOTO	produce a jump to a label expression
LSWITCH	produce a switch jump (list version)
TSWITCH	produce a switch jump (table version)

The actual macro names which appear in an intermediate language program are abbreviations of the names listed above.

2.2 The Machine Description

The machine description is a "program" written in a special-purpose language from which is constructed the machine-dependent tables of the generation phase. The machine description has two functions: (1) it defines the particular abstract machine for which the code generator produces intermediate code, and (2) it specifies the translation from an intermediate language program to the corresponding object language program.

The abstract machine is defined in two sections of the machine description. First, a set of definition statements defines the registers and memory of the abstract machine. Second, in the OPLOC section, the AMOPs are defined in terms of their operand/result locations. The translation from the intermediate language to the object language is specified by a set of macro definitions in the macro section of the machine description. More information on the writing of a machine description may be found in Appendix I; the machine description used in the HIS-6000 implementation is listed in Appendix IV.

2.2.1 Defining the Abstract Machine

In the machine description, the implementer first defines the registers of the abstract machine. For example, the statement

```
regnames (x0,x1,x2,x3,x4,a,q,f);
```

defines the eight abstract machine registers used in the HIS-6000 implementation. The registers X0 through X4 correspond to the first five of eight HIS-6000 index registers, the A and Q correspond to the accumulators, and the F register is a fictitious floating-point accumulator which corresponds to the combined A, Q, and E (exponent) registers on the HIS-6000. The fact that the F register conflicts in the target machine with the A and Q registers is specified by the statement

```
conflict (a,f),(q,f);
```

The remaining HIS-6000 index registers are not represented in the abstract machine since it was not desired that they be used by the code generator in the evaluation of expressions; two of these registers hold "environment pointers," the other is used as a scratch register by some of the macro definitions. There is nothing that requires that the abstract machine registers be implemented as actual machine registers on the target machine; they may also be implemented as fixed memory locations.

For convenience, the abstract machine registers can be gathered into classes; for example, in the HIS-6000 implementation, the statement

```
class x(x0,x1,x2,x3,x4), r(a,q);
```

defines the class of index registers X and the class of general registers R.

The implementer also defines the classes of abstract machine pointers. Pointer classes are necessary on machines which are not byte-addressed since pointers to byte-aligned objects will be handled differently than pointers to word-aligned objects. In the HIS-6000 machine description, the statement

```
pointer p0(1), p1(4);
```

defines the class P0 of byte pointers and the class P1 of word pointers. The "4" indicates that the value of a P1 pointer is always a multiple of four bytes. The fact that there are four bytes per word on the HIS-6000 is specified in the statement

```
size 1(char), 4(int,float), 8(double);
```

A similar statement is used to specify the alignment restrictions.

The statement

```
type int(r), char(r), float(f), double(f), p0(r), p1(x);
```

defines the registers which can hold values of each of the abstract machine data types. For example, in the HIS-6000 implementation, word pointers are held in the index registers X while byte pointers are held in the general registers R.

The definition of the abstract machine is completed in the OPLOC section of the machine description where the implementer specifies the behavior of the abstract machine operations in terms of their operand/result locations. For example, the location definition

```
+d:                 f,M,f;
```

specifies that the AMOP '+d' (double-precision floating-point addition) can take its first operand in the F register and its second operand in any memory location and, under these circumstances, the result is placed in the F register. The construct on the right in the location definition is called an OPLOC; it consists of three location expressions, one for the first operand, second operand, and result (reading from

left to right). A location expression may specify any set of abstract machine registers or any set of memory reference classes; for example, the location expression

```
r | x
```

represents the set consisting of the general registers R and the index registers X, and the location expression

```
~ intlit
```

represents the set consisting of all memory reference classes except that of integer constants. An OPLOC may specify that the result is placed in the first or second operand location. For example, the location definition

```
+i:             r,M,1;
```

specifies that the AMOP '+i' (integer addition) takes its first operand in a general register and its second operand in any memory location, and the result is placed in the register which contained the first operand. This location definition is equivalent to

```
+i:             a,M,a; q,M,q;
```

which explicitly lists the two alternatives. An OPLOC may also specify that the contents of certain registers are destroyed during the execution of an AMOP; for example, the location definition

```
*i:             q,M,q [a];
```

specifies that an integer multiplication destroys the contents of the A register.

2.2.2 Defining the Object Language

The translation from the intermediate language to the object language is specified by a set of macro definitions included in the machine description; macro definitions are provided for the abstract machine instructions and the keyword macros. The simplest form of a macro definition is a single character string which is substituted for the macro call during macro expansion. For example, the macro definition for floating-point unary minus used in the HIS-6000 implementation is

```
-ud:            "       FNEG"
```

This macro definition specifies that each occurrence of a '-ud' abstract machine instruction is to be translated into the assembly language instruction "FNEG" which complements the contents of the F register. The macro definition for '-ud' is closely related to the location definition for '-ud',

```
-ud:            f,,1;
```

which states that the operand is found in the F register and that the result is placed in the F register. A macro definition for an AMOP can assume that the actual operand/result locations appearing in an abstract machine instruction satisfy the constraints specified in the corresponding location definition; at the same time, a macro definition must produce correct code for all combinations of operand/result locations allowed by the location definition.

A macro definition for an abstract machine instruction can refer to symbolic representations of the operation and the operand/result locations by using the character sequences *O (operation), *F (first operand), *S (second operand), and *R (result). These character sequences are abbreviations for calls to an implementer-defined macro which converts an AMOP opcode or a REF into the desired object language

representation. For example, the macro definition for '+i' (integer addition) in the HIS-6000 implementation is

```
+i:             "       AD#R    #S"
```

If the first operand location (which is also the result location) is the A register and the second operand is an external variable "X", then the code produced by this macro definition is

```
        ADA     X
```

which adds the contents of "X" to the A register. A macro definition can also contain character strings whose inclusion in the expansion of a macro call is conditional upon the locations of the operands and/or result. An example is the HIS-6000 macro definition for '<<' (left shift)

```
<<:
(,intlit,):     "       #FLS    %o(#'S)"
(,~intlit,):    "       LXL5    #$
                        #FLS    0,5"
```

which produces different code sequences depending upon whether or not the second operand (the number of bit-positions to shift) is an integer constant. A macro definition may include references to the arguments of the macro call using the character sequences #0, #1, ... #9; a macro definition may include embedded macro calls, such as the "%o(#'S)" in the last example, which returns the value of the integer constant.

A macro definition may also be specified in the form of a C routine. C routine macro definitions are used when processing is needed which is beyond the capabilities of the simple macro scheme so far described. C routine macro definitions may define global variables, perform arithmetic and logical operations, and select code sequences on conditions other than operand locations. In the present implementation, however, C routine macro definitions are unable to interact with the code generation algorithm. In the HIS-6000 implementation, C routine macro definitions are used to translate REFs into GMAP symbols, to translate the source language representations of identifiers and floating-point constants into GMAP, to define character string constants, and to buffer characters while defining storage for variables (GMAP does not have a byte location counter, as is assumed in the intermediate language). The C routine macro definitions used in the HIS-6000 implementation are listed in Appendix V.

3. Generating Code for an Abstract Machine

The most interesting part of the compiler is the code generator since, unlike most code generators which produce code for a fixed target language, the code generator of the C compiler is designed to produce code for a class of abstract machines.

3.1 Functions of the Code Generator

The code generation process consists of three fairly distinct functions. First, there is the generation of intermediate language statements to define and initialize static data areas and constants. Second, there is the translation of source language control structures into labels and branches. Third, there is the translation of source language expressions into sequences of abstract machine operations.

The C compiler is designed to produce assembly language code for conventional machines; thus, the intermediate language statements for defining and initializing static data areas directly correspond to assembly language statements which define symbols, define constants, and align the location counter. The only complication is that the code generator must use the size and alignment information from the machine description in order to specify the sizes and alignments of data areas. More information and redundancy could be added to the intermediate language in order to accomodate a larger class of target languages; see [16] for examples. Another possible improvement would be to emit segment specifying instructions so that the output could be segregated into different segments according to whether it is code, pure data, impure data, or uninitialized data.

The process of translating source language control structures into labels and branches is rather straightfoward. The only complications come when emitting conditional branches which test the value of an expression; these problems are covered in the next section.

3.2 Generating Code for Expressions

The generation of code for expressions is the most difficult part of the problem. The code generator must generate a correct sequence of abstract machine instructions to carry out the indicated operations. The operand and result locations it specifies in the abstract machine instructions must conform to the location definitions provided in the machine description. Moreover, the code generator must keep track of the locations of all intermediate results and correctly administer the abstract machine registers and temporary locations.

The generation of code for expressions is performed in two steps, semantic interpretation and code generation.

3.2.1 Semantic Interpretation

The code generator receives expressions in the form of syntax trees whose interior nodes are source language operators and whose leaf nodes are identifiers and constants. Thus, an expression can be considered to consist of a "top-level" operator along with zero or more operand expressions. The first step in the processing of an expression consists of translating a tree in this form to a more descriptive form whose interior nodes are AMOPs. This translation involves checking the data types of operands, inserting conversion operators where necessary, and choosing the appropriate AMOPs to express the semantics of the source language operators. The selection of an AMOP to replace a source language operator is based primarily on the data types of the operands. For example, on this basis, an addition operator may be translated into either integer addition, double-precision floating-point addition, or one of a number of pointer addition AMOPs. However, it is useful to be able to choose AMOPs also on the basis of what is provided in the machine description. The basic idea is that of defaults. If the semantics of a particular AMOP can be expressed in terms of a composition of more basic AMOPs, then the AMOP can be left undefined in the machine description; the code generator can use the equivalent composition of AMOPs instead. The advantage of having optional AMOPs is that the implementer need define one of

these optional AMOPs in the machine description only if his definition will result in sufficiently better code than will be produced using the equivalent composition of more basic AMOPs.

An example of this technique is the handling of a class of C operators called assignment operators. An example of an assignment operator is '=+', where "L =+ R" is defined to be the same as "L = L + R" except that the expression L is evaluated only once (it may contain side-effects). Consider an expression "L =op R." If the corresponding abstract machine assignment operator is defined in the machine description, then the source language assignment operator is translated into that abstract machine operator; otherwise, the expression "L =op R" is converted to the equivalent form "L = L op R", except that there is only one copy of "L" having two pointers to it (a flag is set in the root node of "L" so that later routines will recognize this fact). Therefore, a particular abstract machine assignment operator need be included in the machine description only if the code sequences it generates are better than the code that would be generated by the equivalent assignment expression. An example from the HIS-6000 implementation is the abstract machine operator '=+i' (integer addition-assignment) which is translated into an add-to-storage instruction. The corresponding floating-point assignment operator '=+d' is not defined in the machine description since no floating-point add-to-storage instruction exists on the machine.

Other examples of optional AMOPs which have been implemented are the pointer comparison operators for pointers other than class P0 pointers (the default is to convert to the "greatest common denominator" pointer class for which the operation is implemented) and the test for null/non-null pointer operators (the default is to convert the pointer to an integer and test for equality/inequality with 0). Other promising candidates for being optional AMOPs are the various increment and decrement AMOPs.

3.2.2 Code Generation

The second step in the processing of an expression is the generation of a sequence of abstract machine instructions to carry out the evaluation of the expression. This code generation is performed by a set of recursive routines, some of which will be described in this section. The operation of the code generation routines is basically top-down. When a call is made to generate code to evaluate an expression, a set of desired locations for the result of that evaluation is also specified. This specification, along with other available information about the operands of the top-level operator of the expression, is used to choose one of the OPLOCs from the top-level operator's location definition in the machine description (location definitions are described in section 2.2.1). From the chosen OPLOC and, possibly, the desired locations for the result of the expression are derived sets of desired locations for the operands of the top-level operator. Recursive calls are then made to generate code to evaluate the operands into these desired locations. Next, an abstract machine instruction is emitted for the top-level operation. Finally, if necessary, abstract machine instructions are emitted to move the result of the expression to an acceptable location.

3.2.2.1 Specifying Desired Locations

A set of desired result locations is specified by a structure called a LOC. A LOC structure has two integer members, LOC.FLAG and LOC.WORD. The possible values of LOC.FLAG are listed below along with their interpretations:

LOC.FLAG	interpretation
0	the "result" is the internal label specified by LOC.WORD (used only for conditional jump AMOPs)
1	the result is to be placed in a register; acceptable registers are specified by one-bits in LOC.WORD (bit 0 corresponds to register number 0, etc.)
2	the result is to be placed in memory; acceptable classes of memory references are specified by one-bits in LOC.WORD (this field is used only to select registers for pointers in indirect references)
3	the result may be left in any location acceptable for values of the particular data type

Note that a particular memory location is never specified as the desired location for a result; rather, classes of possible memory locations are specified.

For convenience, if the LOC passed to the top-level code generation routine specifies that the result is desired in a register, then all registers not capable of containing the particular data type of the expression being evaluated (as defined in the TYPE statement of the machine description) are removed from the LOC. Similarly, if the LOC specifies memory reference classes, then all indirect classes where the pointer register is unable to hold pointers of the corresponding pointer class (as specified by the TYPE statement) are removed from the LOC. Thus where the code generator simply desires that a value be in a register, it may provide a LOC specifying that the result may be left in any register.

The removal of "impossible" registers from a LOC is not performed when such an action would leave no remaining acceptable registers; this situation can actually occur in certain special cases, such as return statements, where an operation requires a value in a register not normally used to hold values of that type.

3.2.2.2 TTEXPR

The top-level code generation routine is TTEXPR. The function of TTEXPR is to generate a sequence of abstract machine instructions which will evaluate a given expression and leave the result in an acceptable location, as specified by a LOC parameter. The operation of TTEXPR begins with the removal of impossible cases from the LOC parameter, as described above. Then, TTEXPR passes the expression and LOC parameters to a routine CGEXPR, which generates abstract machine instructions to evaluate the expression, using the LOC parameter as a non-binding indication of preference. Finally, TTEXPR calls the routine CGMOVE to emit, if necessary, abstract machine instructions to move the result to an acceptable location.

3.2.2.3 CGEXPR

The function of CGEXPR is to generate a sequence of abstract machine instructions which will evaluate a given expression. CGEXPR is given a LOC argument which specifies preferred locations for the result of the expression; however, unlike TTEXPR, this specification is non-binding and is used only where a choice exists.

The operation of CGEXPR consists basically of testing for a set of special cases and then performing the appropriate action, which is usually to call another routine which does the real work. The first special case is where the expression node is shared and the expression has already been evaluated; in this case, no action need be taken. Another special case is where the top-level operator is a conditional AMOP and a value is desired (as opposed to a jump, which is the usual case); in this case, a routine JUMPVAL is called to emit the desired code. The other special cases involve particular top-level operators:

indirection, assignment, conditional expression, function call, and the "leaves" of the expression tree, identifiers and literals; in these cases, the code generation routine corresponding to the particular top-level operator is called. Finally, in all other cases, the routine CGOP is called to emit code to evaluate the expression.

3.2.2.4 CGOP

The function of CGOP is to emit code to evaluate an expression whose top-level operator is not one special-cased by CGEXPR. Like CGEXPR, CGOP is passed a LOC indicating non-binding preferences for the location of the result of the expression.

The operation of CGOP is performed in six steps. First, a routine CHOOSE is called to select an OPLOC from the top-level operator's location definition in the machine description. Second, desired locations for the operands of the top-level operator are determined. Third, a routine EXPR2 is called which makes recursive calls on TTEXPR to emit code to evaluate the operands into the desired locations. Fourth, code is emitted to save any registers which are specified in the machine description to be clobbered by the execution of the top-level operator. Fifth, the exact location of the result of the expression is determined. Sixth, the actual abstract machine instruction for the top-level operator is emitted.

If the result location specified by the LOC parameter is a label, or if the selected OPLOC specifies that the result is left in the first or second operand location, then the exact location of the result of the expression is fixed. Otherwise, a particular register must be chosen from the set of registers specified in the result field of the OPLOC (the compiler is currently unable to handle OPLOCs which specify a set of memory references as the location of the result). In the search for a result register, the priorities are as follows: first, free registers which are preferred result locations; second, busy registers which are preferred result locations; third, free registers which are not preferred result locations; and fourth, busy registers which are not preferred result locations. If a busy register is selected, register contents are saved in temporary locations as necessary.

For the purposes of finding a result register, a register containing an operand is considered free and a register containing a pointer to an operand is given lowest priority. A register containing a pointer to an operand is protected because the implementation of a AMOP may alter the contents of the result register before the operand referenced by the pointer in that register is used. An example is the following HIS-6000 code for the AMOP '+p1' (addition of an integer to a pointer to a word-aligned object):

```
LXL0    I
ADLX0   P
```

This code loads index register 0 with the integer I and then adds to register 0 the pointer P. (The code for the AMOP includes the load instruction since in general integers cannot be stored in the HIS-6000 index registers as they are only halfword registers.) If the code generated for P leaves P referenced through index register 0, the load instruction will "clobber" register 0 before P is accessed by the add instruction:

```
LXL0    I
ADLX0   0,0
```

However, if index register 0 is protected, index register 1 will be chosen instead to hold the result, producing the following correct code:

```
LXL1    I
ADLX1   0,0
```

3.2.2.5 Selecting an OPLOC

The purpose of OPLOC selection is to select a set of operand/result locations for the top-level operator of an expression by choosing one of the OPLOCs from the location definition of the operator in the machine description. The choice of operand/result locations will affect the amount of code produced to evaluate the expression, both because of different code sequences which may be produced by the macro definition for the operator and because of additional loading, storing, and saving operations which may be required in order to set up the operands and move the result to an acceptable location. A general solution, taking into account all possible locations of operands and results, is a complex optimization problem. Instead, a more limited approach has been taken which uses the provided preferences for result locations and available information about the possible result locations of the top-level operators in the operand subexpressions. For example, if an operand is an identifier, then its location is known to be a memory reference of a particular class. Similarly, various operators may be defined in the machine description to always place their result in one of a particular set of registers. Using information of this sort, plus knowledge about the current register usage, a rough estimate can be made of the number of additional load and store instructions which will be required for each OPLOC in the location definition; from the set of OPLOCs, the one with the lowest additional cost is chosen.

For example, consider the expression "I + (J / K)." (For clarity, source language operator symbols are used in this example to represent the corresponding integer abstract machine operations.) Assume the following location definitions (the OPLOCs are numbered for future reference):

+:	r,r,1;	*(1)*
	r,M,1;	*(2)*
	M,r,2;	*(3)*
/:	r1,r,1 [r2];	*(4)*
	r2,r,1 [r3];	*(5)*
	r3,r,1 [r4];	*(6)*
	r1,M,1 [r2];	*(7)*
	r2,M,1 [r3];	*(8)*
	r3,M,1 [r4];	*(9)*

Here M represents all memory reference classes and r represents a set of general registers consisting of r1, r2, r3, and r4. The division operator is modeling a machine instruction which produces pairs of results (the quotient and remainder) in adjacent registers. For the division abstract machine operator, only the quotient is used; the other register is considered to be "clobbered" by the execution of the operator. Note that one can deduce from these location definitions that both operators always leave their results in general registers.

The generation of code for the expression "I + (J / K)" begins with the selection of an OPLOC from the location definition of the '+' operator. In this case, all of the OPLOCs specify the same set of result locations (the general registers); thus, the desired locations for the result of the expression does not affect the choice of OPLOCs. Instead, the choice is made on the basis of the possible locations for the operands. In this case, the first operand is a variable I which is known to be a memory reference of a particular class. The second operand is the result of a division operator which is known to leave its results in either r1, r2, or r3. On this basis, OPLOC *(3)* is chosen because no extra operations are needed to move the operands into acceptable locations, whereas both OPLOCs *(1)* and *(2)* do require such extra operations.

Next, a recursive call is made to generate code to evaluate the subexpression "J / K." The desired locations for the result of this expression are those specified by the chosen '+' OPLOC for its second operand, namely r, the set of general registers. However, since the '+' OPLOC specifies that the second operand location is also the location of the result of the '+' operator, the intersection of that location set with the set of desired locations for the result of the '+' operator is used instead, if that intersection is

non-null. Thus, the following factors are used in selecting an OPLOC for the '/' operator: first, which of the possible result registers (r1, r2, r3) are desired result locations; second, which of the possible result registers are free; and third, which of the "clobbered" registers (r2, r3, r4) are free. In this particular situation, the possible location of the first operand (J) is a memory reference and thus does not favor any of the OPLOCs. However, the second operand, which is also known to be a memory reference, favors OPLOCs *(7)*, *(8)*, and *(9)*.

In addition, when selecting an OPLOC from a location definition, certain OPLOCs may be rejected entirely because they specify conditions which can not be met. For example, if an OPLOC specifies (either directly or indirectly through an operand location) that the result is left in a register, but the result is desired in memory, then that OPLOC will be rejected if a temporary location is not acceptable. The OPLOC is rejected because, given a value in a register, the only general method by which the code generator can make that value into a memory reference is by saving it in a newly allocated temporary location. (Recall that a specific memory location is not provided for the result, only a set of acceptable memory reference classes.) Similarly, if the result will be in memory and is desired in memory, then that OPLOC will be rejected if there are one or more possible result memory reference classes which are not acceptable result locations; this is done because the code generator is not capable of transforming a memory reference from one class to another. Similar checking is performed on the operand location specifications in the OPLOC: if an operand is required by the OPLOC to be in memory but not all non-indirect memory reference classes are allowed, then that OPLOC will be rejected if the operand operator is not guaranteed to place its result in an acceptable memory location or if it can place its result in a register but temporary locations are not acceptable. These restrictions allow a location definition to contain extra OPLOCs which apply only in special cases since such OPLOCs will never be chosen unless the special cases hold.

An example of how the OPLOC selection method can be utilized in the writing of a machine description is the following definition of the '+p1' AMOP (addition of a integer to a pointer to a word-aligned object) taken from a hypothetical HIS-6000 machine description (the described OPLOC selection method was not implemented at the time the actual HIS-6000 machine description was written). The shortest code for executing the '+p1' operation in the general case is

```
LXL0    I
ADLX0   P
```

where I is the integer in the low-order half of a word in memory and P is the pointer in the high-order half of a word in memory. The result of this operation is left in an index register; thus the OPLOC for this code sequence is

```
M,M,x;
```

However, if both the integer and the pointer must be computed into registers (which occurs frequently in referencing elements of an array), the integer and the pointer must first be stored into temporary locations before this code sequence can be applied. Therefore, using the given code sequence under these circumstances results in excessive object code. The desired code is

```
ALS     18
STA     TEMP
ADLX0   TEMP
```

which shifts the integer in the general register into the high-order halfword, stores it into a temporary location, and adds it to the pointer in the index register. The OPLOC for this code sequence is

```
x,r,1;
```

In the case where the pointer is in an index register and the integer is a constant "n", then the desired code is

```
EAX0    n,0
```

with an OPLOC of

```
x,intlit,1;
```

The described OPLOC selection method allows all three OPLOCs to be included in the location definition for '+p1'. In particular, it guarantees that the third OPLOC will never be selected unless the second operand is an integer constant.

3.2.2.6 Generating Code for Subexpressions

After an OPLOC has been selected, CGOP calls a routine EXPR2 to make recursive calls on TTEXPR to generate code to evaluate the operands of the top-level abstract machine operator. The LOC arguments passed to TTEXPR in these calls are taken from the operand fields of the selected OPLOC and, in the case of operators which place their result in an operand location, the desired locations for the result of the top-level operator. If there are two operands, EXPR2 makes sure that the two operands will not require the use of the same register (for example, by using a register to hold both one operand and a pointer to the other operand); this is done by checking the LOCs for "overlap" and removing certain possibilities. In addition, EXPR2 evaluates first the operand which is more complicated on the basis of the sizes of the subtrees for the two operands; this tends to reduce the number of saving and restoring operations performed. In the course of generating code to evaluate an operand of a binary abstract machine operator, it may be necessary to use the register containing the already computed value of the other operand or a pointer used to reference it, in which case code is generated to save the contents of this register in a temporary location. Thus, after generating code to evaluate both operands, EXPR2 calls a routine RESTORE to generate code, if necessary, to restore the saved value to its original register.

3.2.2.7 Register Management

The status of the various abstract machine registers with regard to register allocation is contained in an array of structures called REGTAB. Each element structure of the array represents the current state of one abstract machine register. An element structure consists of two members: UCODE, an integer indicating the current use of the register, and REP, a pointer to the subexpression tree whose value is currently in the register. The possible values of UCODE are listed below with their interpretations:

UCODE	Interpretation
0	the register is free
-1	the register contains the value of the expression pointed to by REP
-2	the register has been marked "do not use unless necessary" for the purpose of finding a register for the result of an AMOP; although the register contains a pointer to one of the operands of the AMOP, it is free in that it may be selected as a last resort without having to save its contents.
n>0	the register does not directly contain a value, but there are "n" conflicting registers containing values which must be saved before this register can be used.

The routines used in register management are described below:

CLEAR(R)	- Register R, which must directly contain the value of an expression, is made available for use; its current value is not saved.
ECLEAR(E)	- The register associated with the expression E, if any, is CLEARed.
FREEREG(W)	- A register from the set specified by W is made available for use; the contents of registers are saved if necessary.
GETREG(W1,W2)	- If possible, an unmarked register from the set W1 is made available for use. Otherwise, if possible, an unmarked register from the set W2 is made available for use. Otherwise, a marked register from the set W1 is made available for use. Within each set, free registers are chosen in preference to busy registers; if a busy register is chosen, its contents are saved.
MARK(E)	- If the expression E is an indirect reference, the register containing the pointer is marked "do not use unless necessary."
NBUSY(W)	- Return the number of busy registers in the set W.
NFREE(W)	- Return the number of free registers in the set W.
RESERVE(R,E)	- Register R is allocated to hold the value of the expression E. Register R must be available for use.
RESTORE(E)	- If the value of the expression E (or a pointer in the case of an indirect reference) has been saved in a temporary location, it is restored to the original register.
SAVE(R)	- Register R is made available for use by saving the contents of whatever registers are necessary.
UNMARK(E)	- Undo a MARK.

The following is a typical series of calls made by CGOP in the generation of code for an expression E whose top-level operator is a binary operator with operands OP1 and OP2:

OPLOC=CHOOSE(E,LOC)	choose an OPLOC
EXPR2(OP1,OP2)	recursively generate code to evaluate the operands into acceptable locations
ECLEAR(OP1) ECLEAR(OP2)	make operand registers available for the result
SAVE(*)	save "clobbered" registers, if any
MARK(OP1) MARK(OP2)	mark registers used to hold pointers to operands
R=GETREG(*,*)	select a result register
UNMARK(OP1) UNMARK(OP2)	unmark any marked registers
RESERVE(R,E)	reserve result register

3.2.2.8 Possibilities for Failure

The code generator can fail in two ways: (1) it can reach an impossible situation and announce a compiler error, and (2) it can unknowingly generate incorrect code. Examples of impossible situations are (1) discovering that there are no acceptable OPLOCs in the location definition for an operator, (2) being told that the result must be placed in a register from the empty set of registers, and (3) discovering that an essential location definition or macro definition of an abstract machine operator was not provided by the implementer. The most likely cause of a failure is an incorrect machine description. Examples of errors

which can be made in the machine description are (1) an OPLOC specifying that both operands must be in the same register, (2) an OPLOC specifying a set of memory reference classes for the result location, (3) a macro definition containing errors, and (4) a macro definition which does not anticipate a particular operand or result location, or combination thereof, allowed by the location definition or otherwise essential (in the case of move operations which must be capable of moving among registers and between registers and memory). Some of these errors could be detected by the program which processes the machine description (GT). Another possible cause of failure is an abstract machine with an insufficient number of registers. Such a machine may require that a register be used to hold both a pointer to an operand and the result of an operation; as described above, this situation may result in incorrect code. Hopefully, abstract machine models of real machines will not suffer from this problem. Of course, the other possible cause of failure is a bug in the code generator itself. It would be interesting and useful if such a code generation algorithm could be proven correct, given sensible restrictions on the machine description and the assumption of correct macro definitions.

4. Conclusions

This paper has described the implementation of a portable compiler for the programming language C. The compiler was first implemented by the author in a seven month period on the Bell Laboratories Computer Science Research Center's PDP-11/45 UNIX system. The compiler was then used to compile itself, and the resulting code moved to the HIS-6000. Another month was spent debugging the compiler until the version of the compiler compiled on the HIS-6000 successfully compiled itself. This was regarded as a significant test of the compiler.

4.1 The Compiler

The major problem with the compiler itself is its speed. The compiler appears to be more than twice as slow as other compilers for similar source languages. This slowness is due almost entirely to the use of a macro expansion phase (a phase not likely to be present in ordinary compilers), since the compiler tends to spend half or more of its time in the macro expansion phase. The slowness of the compiler seems to be a problem inherent in the chosen compiler structure; no amount of mere recoding is likely to significantly reduce the percentage of time spent in the macro expansion phase. One approach toward improving the speed of the compiler would be to eliminate non-essential processing such as the construction and interpretation of character-string representations of macro calls and the rescanning of macro definitions. The macro language could be modified so that the result of the expansion of a macro call would never be needed as an argument to another macro call and thus could be printed directly, rather than returned as a string and rescanned. Given this restriction, the macro definitions could be compiled into procedures which simply print strings and call other procedures. These procedures could be called directly by the code generator; alternatively, they could be called by a procedure which interprets a suitable encoding of the intermediate language.

A second problem with the compiler is its size, in terms of both the amount of file space necessary to support an implementation of the compiler and the amount of memory required to execute the compiler phases. The source of the compiler is about 250K characters, the source of GT is about 80K characters; thus, the file space required for source, object libraries, and executable files is on the order of 1M characters. Only the size of the code of the code generator is a result of designing the compiler to be portable; it is likely that a code generator designed for a specific machine would be much smaller. Other reasons for the large size of the compiler stem from the particular programming techniques used. In particular, keeping the entire tree representation of a function in core at one time during code generation requires that a large block of storage be reserved. Also, the use of a bottom-up table-driven LALR(1) parser seems to result in a larger syntax analysis phase than would result from using recursive descent, as does the UNIX C compiler. The large size of the compiler limits the number of computer systems which can support the compiler.

Despite these problems, it is believed that were one prepared to make the investment necessary to implement C on another machine, the size difficulties and related costs would be outweighed by the relative speed with which one could bring up a working implementation. One could then concentrate on making it more efficient, having the advantages of a C compiler to work with and the ability to program in C.

The least flexible machine-dependent component of the compiler is the code generation algorithm. It is acknowledged that a clean mechanism for allowing the implementer to tailor the code generation algorithm through the addition of procedural knowledge would be an improvement. On the other hand, clinging to the idea that the code of the compiler will never be touched is unrealistic. A likely prospect for modification is the code related to the calling sequence since it may be desired to use a system standard calling sequence instead of the one built into the compiler. Another problem which would be solved most easily by modifying the code generator is the IBM S/360 addressing problem. Because a S/360 instruction cannot contain an arbitrary memory address, C external variables must be referenced by first loading a register with a pointer to the variable (an address constant) and then using the register as a base register in the actual instruction. These actions could be performed by the macro definitions using

conditional expansion; however, it would be easier to modify the code generator to handle this particular case.

The most direct method of moving a portable compiler based on a machine description requires access to an existing implementation of the compiler. The process of moving a compiler written in its own language from machine A to machine B is as follows: First, one writes a machine description for machine B. Second, the machine description is used by a construction program running on machine A to produce a new compiler which produces code for machine B. Third, the compiler on machine A is used to compile the new compiler, producing a compiler which runs on machine A but produces code for machine B. Fourth, the new compiler is used to compile itself, producing a compiler which runs on machine B and produces code for machine B. This process is called a half bootstrap. On the other hand, the Poole and Waite approach does not require the use of an existing implementation. One need write only an interpreter or a translator for a very simple abstract machine language in order to move a program to a new machine. This technique is called a full bootstrap. In practice, the need for a half bootstrap often represents a significant obstacle to moving a program.

The full bootstrap method can be used to move a portable compiler based on a machine description as follows: Initially, a simple imaginary machine is defined as a vehicle for bootstrapping. A compiler which runs on and produces code for this imaginary machine is then constructed using the half bootstrap method described above. Now, in order to move the compiler to a new machine, one implements an interpreter for the imaginary machine on the new machine. This action results in an "existing implementation" of the compiler, running on the new machine, which can then be used to carry out the half bootstrap as described above.

4.2 The Compiled Code

Although there are weak spots, the code produced by the compiler is good considering that it is almost completely unoptimized. It is certainly better than would be produced if the abstract machine were the typical machine-independent abstract machine with one accumulator and one index register, given the same complexity of the macro definitions (they do not perform register allocation). Such an implementation would not be able to take advantage of the HIS-6000's two accumulators or the multiple index registers, nor would it recognize the fact that byte pointers cannot fit in the index registers.

One of the weak spots in the compiled code concerns floating-point operations. The code generator "performs" all floating-point operations in double-precision, issuing single-to-double conversion operations before using single-precision operands. It is unable to utilize the HIS-6000 machine instructions which operate on a single-precision operand in memory and a double-precision operand in the F register. Since the implementation of a single-to-double conversion is to load the single-precision operand into the F register, very poor code is produced for single-precision floating-point expressions (as opposed to very good code for double-precision expressions). One way to handle this situation would be to implement a general subtree-matching facility for optimization. With such a facility, the implementer specifies in the machine description that a particular combination of abstract machine operators (specified in the form of a tree) is to be replaced by the code generator with a new abstract machine operator; the new operator is defined by the implementer in the machine description just like any of the built-in operators. In the floating-point case, one would specify that a subtree of the form (using a LISP-like notation)

```
( double-prec-add ( *1 , single-to-double ( *2 ) ) )
```

would be replaced by

```
( single-prec-add ( *1 , *2 ) )
```

where single-prec-add is a new abstract machine operator which would be defined to be the "FAD" instruction. This method of subtree-matching can be compared to the hierarchy of abstract machines

method in that the new abstract machine operators can be considered to be instructions of a higher-level abstract machine. The differences are that, in the case of the subtree-matching method, the definition of higher-level operators is optional (thus there is no multistage translation when optimization is not desired or needed) and that the implementer defines the higher-level operators to suit his needs. The subtree-matching approach to machine-dependent code optimization has been investigated by Wasilew [17].

Another weakness in the compiled code concerns array subscripting. Instead of placing the offset of an array element into an index register and performing an indexed memory reference, the code generator adds the offset to a pointer to the base of the array, producing a pointer (in an index register) which is then used to reference the array element. Thus, the code generator regards index registers only as base registers to hold pointers, and not as index registers to hold offsets. One reason for not implementing the capability of using index registers for subscripting is that this method of subscripting is often not possible. For example, on machines like the HIS-6000 with single-indexed instructions, this method can be used only for external and static arrays; all other arrays require the use of an index register just to reference the base of the array. (Actually, one can perform double-indexing on the HIS-6000 by using an indirect word; however, this was not recognized at the time the compiler was written.) The capability of using index registers for subscripting could be implemented using the subtree-matching facility described above; one would test for subtrees of the form

(pointer-add (address-of (extern | static), <any>))

and replace them with a new abstract machine operator which would be defined to produce the desired code. A more satisfying solution would give the code generator more knowledge about addressability so that it could use index registers for subscripting whenever possible, based on information given in the machine description.

A third weakness of the compiled code is the use of indirection. The code generator only indirects through pointers in registers; it is unable to utilize an indirection-through-memory facility (except through a specific location which implements an abstract machine register). Again, a better understanding of addressing is what is really needed.

4.3 Summary of Results

This paper has presented a technique for the design of portable compilers and has demonstrated its practicality through the implementation of a portable C compiler. The main difference between this work and the previous work described in section 1.2 is that in this work, the system was designed specifically for the language being implemented; it is this restriction which contributes most to the practicality of the approach. In addition, this work has emphasized the concept of a machine-dependent abstract machine, thus tying together the work on portable compilers and program transferability.

The advantages of the technique presented in this paper over the technique of rewriting some or all of the generation phase are (1) that the implementer can modify the compiler to produce code for a new machine with less effort and in less time, and (2) that the implementer can be more confident in the correctness of the modifications. Almost the entire code of the generation phase, already tested in the initial implementation, is unchanged in the new implementation. This code includes the code generation algorithm, the register management routines, and the macro expander. Furthermore, the modifications which must be made are localized in two areas, the machine description and the C routine macro definitions. The implementer is primarily concerned with the correct implementation of the individual abstract machine instructions. The interaction among these instructions, in terms of their correct ordering and the use of registers and temporary locations, is handled by the code generation algorithm and need not be of concern to the implementer. It is this reduction in the complexity of the problem which leads to the increased confidence in the results of the modification.

The portability of the compiler has been tested by the construction of version of the compiler for the DEC PDP-10. The initial machine description and macro definitions for the PDP-10 implementation were written and debugged by the author in a period of two days.

4.4 Further Work

There are three main directions for further work. One is to develop machine models which will allow the generation of acceptable code for a larger class of machines. Such machine models will have the effect of reducing the complexity of the descriptions of machines which do not completely correspond to the machine model described in this paper. With the HIS-6000, for example, the only major area of complexity in the machine description is that of character manipulation. One would desire a machine model which allows the implementer to describe more conveniently the implementation of characters on his machine. Similarly, a machine model which allows a better understanding of addressing would be desirable.

Another direction for further work is to develop machine-independent code generation algorithms which will produce more efficient code. In particular, the problem of register allocation under complex constraints should be examined. In addition, techniques for allowing the implementer to extend easily and safely the code generation algorithm through the addition of procedural knowledge should be developed. Such techniques should allow the compiler to be modified to produce code for unanticipated new machines.

The third direction for further work is to apply the technique of portable compilers to more complicated and more powerful languages. The technique of using a machine-independent code generation algorithm and a machine description, even aside from portability, results in a very clean and modular code generator. It would be interesting to see if this technique could reduce the complexity of code generators for large languages and whether portability could still be obtained without destroying the efficiency of the object code.

References

1. Ritchie, D. M, C Reference Manual, Bell Laboratories internal memorandum.

2. Snyder, A., C Reference Manual, Bell Laboratories internal memorandum.

3. Richards, M, "BCPL: A Tool for Compiler Writing and System Programming," Proc. SJCC 1969, pp. 557-566.

4. Strong, J., et. al., "The Problem of Programming Communication with Changing Machines -- A Proposed Solution," Comm. ACM 1:8 (Aug. 1958) pp. 12-18, 1:9 (Sept. 1958) pp. 9-15.

5. Feldman, J. and Gries, D., "Translator Writing Systems," Comm. ACM 11:2 (Feb. 1968), pp. 77-113.

6. Feldman, J. A., "A Formal Semantics for Computer Languages and Its Application in a Compiler-Compiler," Comm. ACM 9:1 (Jan. 1966), pp. 3-9.

7. Englund, D. and Clark, E., "The CLIP Translator," Comm. ACM 4:1 (Jan. 1961), pp. 19-22.

8. Halstead, M. H., Machine-Independent Computer Programming, Spartan Books, Washington 1962.

9. Richards, M., "The Portability of the BCPL Compiler," Software Practice and Experience 1:2 (1971), pp. 135-146.

10. Poole, P. C. and Waite, W. M, "Portability and Adaptability," Advanced Course on Software Engineering, Springer-Verlag, Berlin 1973, pp. 183-277.

11. Poole, P. C. and Waite, W. M., "Machine Independent Software," Proc. ACM Second Symposium on Operating Systems Principles.

12. Brown, P. J., "Levels of Language for Portable Software," Comm. ACM 15:12 (Dec. 72), pp. 1059-1062.

13. Sibley, R. A., "The SLANG System," Comm. ACM 4:1 (Jan. 1961), pp. 75-84.

14. Miller, P. L., Automatic Creation of A Code Generator from a Machine Description, M.I.T. Project MAC Technical Report TR-85, 1971.

15. Aho, A. V. and Johnson, S. C., "LR Parsing," Computing Surveys 6:2 (June 1974), pp. 99-124.

16. Coleman, S. S., Poole, P. C., and Waite, W. M., "The Mobile Programming System, JANUS," Software Practice and Experience 4:1 (1974), pp. 5-23.

17. Wasilew, S. G., A Compiler Writing System with Optimization Capabilities for Complex Object Structures, Ph.D. Thesis, Northwestern University, Evanston, Illinois 1971.

18. Johnson, S. C., Bell Laboratories internal document.

```
$          data       cz,copy
 t4 . $ot $cs $sy $er $ma $st $hm >>$el
$          endcopy
$          break
$          program    rlhs,on1
$          limits     ,18k,,1000
$          prmfl      h*,r,r,sny/bt5
$          prmfl      el,r/w,,#/%.e
$          file       er,e1r,5l
$          file       cs,c1r,5l
$          data       cz,copy
 bt5 . $er $cs >>$el
$          endcopy
$          endjob
```

Appendix I - The Machine Description

The format of the machine description is described in detail in the following sections. Examples are taken from the HIS-6000 machine description given in Appendix IV in an attempt to explain the process of writing a machine description which will result in the desired code being produced by the code generator. The convention of writing syntactic alternatives on separate lines is used throughout.

1. Definition Statements

The machine description begins with a series of definition statements. These definition statements are described in the sections below in the order in which they should appear in the machine description.

1.1 The TYPENAMES Statement

The TYPENAMES statement defines the names which are used in the machine description to represent the primitive C data types: character, integer, floating-point, and double-precision floating-point. The form of the TYPENAMES statement is

```
<typenames_stmt>:    typenames ( <name_list> ) ;
<name_list>:         <name_list> , <name>
                     <name>
```

The first name corresponds to the internal type number 0, the second with type 1, etc. Because the internal type numbers are fixed in the compiler, the TYPENAMES statement should always be (equivalent to)

```
typenames (char, int, float, double);
```

1.2 The REGNAMES Statement

The REGNAMES statement defines the names of the abstract machine registers; these registers are assigned internal register numbers (used in REF.BASE, section 2.1.1.2), starting with register number 0, in the order in which they appear in the REGNAMES statement. The form of the REGNAMES statement is similar to that of the TYPENAMES statement; for example, the REGNAMES statement used in the HIS-6000 implementation is

```
regnames (x0, x1, x2, x3, x4, x5; a, q, f);
```

In this example, all but the F register correspond directly to actual registers on the HIS-6000: registers X0 through X4 are the first five (out of eight) index registers, registers A and Q are the two accumulators. The F register is a fictitious floating-point accumulator which in reality corresponds to the combined A, Q, and E (exponent) registers. The fact that the F register conflicts with the A and Q registers is specified in the CONFLICT statement, described below. Only those actual machine registers which are to be used by the code generator in producing code to evaluate expressions should be included in the REGNAMES statement; registers used only for environment pointers, auxiliary address calculations, or other scratch calculations performed within the code for a single AMOP should not be included in the REGNAMES statement. For example, on the HIS-6000, three index registers are not defined in the REGNAMES statement: X7, which contains a pointer to the current stack frame, X6, which contains a pointer to the current argument list, and X5, which is used as a scratch register by AMOPs which access characters.

1.3 The MEMNAMES Statement

The MEMNAMES statement associates names with the various classes of memory references as specified by negative values of REF.BASE (section 2.1.1.2). The form of the MEMNAMES statement is similar to that of the TYPENAMES statement; for example, the MEMNAMES statement used in the HIS-6000 implementation is

```
memnames (reg, auto, ext, stat, param, label, intlit, floatlit, stringlit, ix0, ix1, ix2, ix3, ix4, ia, iq);
```

The first nine names refer to predefined memory reference classes (REF.BASE = 0,-1,-2, ... ,-8), the remaining names refer to indirect references through the abstract machine registers defined in the REGNAMES statement (REF.BASE = -9,-10, ...). The first name "reg" is never used; it serves only as a placeholder. No name is provided for indirect references through the F register since the F register is not used to hold pointers and, being the highest numbered register, omitting it does not affect the positions of the other names in the list.

1.4 The SIZE Statement

The SIZE statement defines the sizes of the primitive C data types in terms of bytes. The form of the SIZE statement is

```
<size_stmt>:        size <size_def_list> ;
<size_def_list>:    <size_def_list> , <size_def>
                    <size_def>
<size_def>:         <integer> ( <type_list> )
<type_list>:        <type_list> , <type>
                    <type>
```

The integers specify sizes in bytes; the types are the names of primitive C data types (as specified in the TYPENAMES statement) with the corresponding size. For example, the SIZE statement used in the HIS-6000 implementation is

```
size 1(char),4(int,float),8(double);
```

All addresses computed by the compiler are in terms of byte addressing; byte addresses are converted to word addresses for non-character operations by the macro definitions. For example, on the HIS-6000, if the first element of an integer array begins at offset 0 in the static area, then subsequent elements of the array are at offsets 4, 8, 12, 16, etc.

1.5 The ALIGN Statement

The ALIGN statement defines the alignment factors of the primitive C data types; these alignment factors are in bytes. The (byte) address of a variable with an alignment factor "n" must be zero modulo "n"; for example, on the HIS-6000, the (byte) address of an integer must be a multiple of 4. An alignment factor must be divisible by all smaller alignment factors; this allows the compiler to assign addresses relative to a base which satisfies the highest alignment restriction. The form of the ALIGN statement is similar to that of the SIZE statement; for example, the ALIGN statement used in the HIS-6000 implementation is

```
align 1(char),4(int,float),8(double);
```

1.6 The CLASS Statement

The CLASS statement is an optional statement which allows the implementer to define classes of abstract machine registers which are used in similar ways; the register classes so defined can then be used in the machine description as abbreviations for the corresponding lists of registers. The form of the CLASS statement is

```
<class_stmt>:          class <class_def_list> ;
<class_def_list>:      <class_def_list> , <class_def>
                       <class_def>
<class_def>:           <name> ( <register_list> )
<register_list>:       <register_list> , <register>
                       <register>
```

The name is the name of the register class, the registers are the names of the abstract machine registers (as specified in the REGNAMES statement) which make up the corresponding register class. The CLASS statement used in the HIS-6000 implementation is

```
class x(x0,x1,x2,x3,x4), r(a,q);
```

This statement defines the class of index registers X and the class of general registers R.

1.7 The CONFLICT Statement

The CONFLICT statement is an optional statement which allows the implementer to specify abstract machine registers which conflict in the actual implementation. The form of the CONFLICT statement is

```
<conflict_stmt>:       conflict <conflict_def_list> ;
<conflict_def_list>:   <conflict_def_list> , <conflict_def>
                       <conflict_def>
<conflict_def>:        ( <register> , <register> )
```

Each register pair specifies two abstract machine registers such that only one of the registers can be in use at one time. The CONFLICT statement used in the HIS-6000 implementation is

```
conflict (a,f), (q,f);
```

which indicates that the F register conflicts with both the A and Q registers.

1.8 The SAVEAREASIZE Statement

The SAVEAREASIZE statement is used to specify the size of the save area which is reserved at the beginning of each stack frame. The save area is generally used for saving registers upon entry to a function, for chaining stack frames together, and for holding other per-invocation information. The form of the SAVEAREASIZE statement is

```
saveareasize <integer> ;
```

The integer specifies the size (in bytes) of the save area. The save area used in the HIS-6000 implementation is 16 bytes (4 words) long.

1.9 The POINTER Statement

The POINTER statement defines classes of pointers according to their resolution; these pointer classes represent different implementations of pointers on the target machine. The resolution of a pointer corresponds to the alignment factors of the objects to which it can refer; in particular, a pointer with a resolution of "n" bytes can refer only to objects whose alignment factors are multiples of "n" bytes. The primary use of pointer classes is on machines whose smallest addressable unit is larger than bytes; in this case, two pointer classes are defined: one which can resolve only machine-addressable units and another which can resolve individual bytes. By defining separate pointer classes, the implementer allows computations involving pointers which are known to refer to machine-addressable units to be performed in terms of machine-addressable units, and therefore more efficiently. The form of the POINTER statement is

<pointer_stmt>:	pointer <pointer_def_list> ;
<pointer_def_list>:	<pointer_def_list> , <pointer_def> <pointer_def>
<pointer_def>:	<name> (<integer>)

The names define the names of the pointer classes, the integers are the resolutions of the corresponding pointer classes. At least one and no more than four pointer classes may be defined; these pointer classes are referred to as P0, P1, P2, and P3 in the specification of the AMDPs.

The POINTER statement used in the HIS-6000 implementation is

```
pointer p0(1), p1(4);
```

P0 is the class of pointers to byte-aligned objects; P1 is the class of pointers to word-aligned objects. Word pointers can be held and operated upon in the index registers; byte pointers are operated upon in the general registers and indirected through by subroutine.

1.10 The OFFSETRANGE Statement

The OFFSETRANGE statement is an optional statement which defines, for each pointer class defined in the POINTER statement, the range of offsets permitted in references indirect via such a pointer (see section 2.1.1.2). The form of the OFFSETRANGE statement is

<offsetrange_stmt>:	offsetrange <offset_def_list> ;
<offset_def_list>:	<offset_def_list> , <offset_def> <offset_def>
<offset_def>:	<pointer_class_name> (<lo_bound> , <hi_bound>)

where the lo_bounds and hi_bounds are optional integers. Each offset_def specifies the range of allowable offsets for a particular pointer class; this range is the set of integers not less than lo_bound and not greater than hi_bound. If a bound is not present, then the range is considered unbounded in the corresponding direction. If no range is specified for a pointer class, then only zero offsets are allowed; any specified range must include zero.

1.11 The RETURNREG Statement

The RETURNREG statement specifies in which registers functions returning values of various types return those values. Registers must be specified for types INT and DOUBLE as well as for all pointer classes defined in the POINTER statement. The form of the RETURNREG statement is

<returnreg_stmt>:	returnreg <return_def_list> ;
<return_def_list>:	<return_def_list> , <return_def> <return_def>
<return_def>:	<register> (<type_list>)

The types may be names of primitive C data types as defined in the TYPENAMES statement or names of pointer classes as defined in the POINTER statement; the corresponding register is defined to be the register in which functions returning values of those types will place the returned values. For example, the RETURNREG statement used in the HIS-6000 implementation is

```
returnreg q(int,p0,p1), f(double);
```

It is advised that pointers of all classes be returned in the same register in a compatible form to avoid errors caused by mismatches in the declarations of functions returning pointers.

1.12 The TYPE Statement

The TYPE statement defines which registers are to be used in the evaluation of expressions to hold values of the various abstract machine data types. The form of the TYPE statement is

```
<type_stmt>:        type <type_def_list> ;
<type_def_list>:    <type_def_list> , <type_def>
                    <type_def>
<type_def>:         <type> ( <register_list> )
```

The type is the name of a primitive C data type as defined in the TYPENAMES statement or the name of a pointer class as defined in the POINTER statement; the registers are the abstract machine registers or classes of abstract machine registers which may be used to hold values of the corresponding type. For example, the TYPE statement used in the HLS-6000 implementation is

```
type char(r),int(r),float(f),double(f),p0(r),p1(x);
```

The registers specified in the TYPE statement need not include every register physically capable of holding a particular type; only those registers which the implementer desires to use in evaluating expressions of that type should be included in the TYPE statement. In the HLS-6000 example, only the index registers (X) are specified for the pointer class P1 even though the general registers (R) are capable of holding such pointers and, in fact, a general register (the Q register) is used to hold such a pointer when returned by a function call; this was done in order to minimize unnecessary use of the general registers which are relatively few in number.

2. The OPLOC Section

In the OPLOC section of the machine description, the AMOPs are defined in terms of the possible locations of their operands and the corresponding locations of their results. Each definition consists of a list of triples called OPLOCs; an OPLOC specifies a particular set of first operand locations, second operand locations, and result locations. An OPLOC may also specify that one or more registers are clobbered by the execution of the code for an abstract machine instruction; this informs the code generator that it may be necessary to emit instructions to save the contents of the clobbered registers before emitting the abstract machine instruction. The forms of an OPLOC are

```
<loc_expr> , <loc_expr> , <loc_expr> ;
```

and

```
<loc_expr> , <loc_expr> , <loc_expr> <clobber> ;
```

where a clobber is a list of one or more register names separated by commas and enclosed in square brackets. The location expressions specify locations for the first operand, second operand, and result, respectively. A location expression specifies either a set of registers or a set of memory reference classes; these sets may be specified using particular registers or memory reference classes along with the operations of union ('|') and negation ('~'). The syntax of a location expression is

```
<loc_expr>:          <register_expr>
                     <memory_expr>
                     1
                     2
                     <null>

<register_expr>:     <register_expr> | <register_expr>
                     ~ <register_expr>
                     ( <register_expr> )
                     <register_name>
                     <register_class_name>

<memory_expr>:       <memory_expr> | <memory_expr>
                     ~ <memory_expr>
                     ( <memory_expr> )
                     <memory_ref_class_name>
                     M
                     indirect
```

The negation operator '~' has precedence over the union operator '|'. The location expressions "1" and "2" may be used only for the location of a result; they specify that the result is placed in the first or second operand location, respectively. Only the location expression for the second operand of a unary AMOP may be null. The location expression "M" represents the set of all memory reference classes; the location expression "indirect" represents the set of all indirect memory reference classes.

The OPLOCs are associated with AMOPs in location definitions which consist of one or more AMOP labels followed by one or more OPLOCs:

```
<loc_def>:           <AMOP_list> <OPLOC_list>
<AMOP_list>:         <AMOP_list> <AMOP_label>
                     <AMOP_label>
<AMOP_label>:        <AMOP> :
<OPLOC_list>:        <OPLOC_list> <OPLOC>
                     <OPLOC>
```

Each AMOP in the list of AMOP labels is associated with the list of OPLOCs; each OPLOC in the list of OPLOCs represents an acceptable set of operand/result locations for each of the AMOPs. For example, the location definition

```
+d: -d: *d: /d:      f,M,f;
```

used in the HIS-6000 machine description specifies that the AMOPs for double-precision floating-point addition, subtraction, multiplication, and division all take their first operand in the F register, their second operand in memory, and place their result in the F register. Another example is the location definition

```
=<<: =>>:            M,a,q; M,q,a;
```

which specifies that the AMOPs left-shift-assignment and right-shift-assignment both take their first operand in memory, their second operand in a general register, and place their result in the other general register. A third example is the location definition

```
*i: /i:              q,M,q[a];
```

which specifies that the AMOPs for integer multiplication and division both take their first operand in the Q register, their second operand in memory, place their result in the Q register, and clobber the contents of the A register in the process. Note that the location definitions

```
+i:             r,M,l;
```

and

```
+i:             r,M,r;
```

are not equivalent. The second definition allows the code generator to emit an abstract machine instruction which adds an integer in memory to an integer in the A register and places the result in the Q register; the first definition requires that the result be placed in the register containing the first operand.

The OPLOC section of the machine description consists of a sequence of location definitions which define the AMOPs of the intermediate language. (A small number of AMOPs should not be defined in the OPLOC section of the machine description; these are indicated in Appendix II.) An AMOP may appear no more than once in the OPLOC section of the machine description.

3. The Macro Section

The macro section of the machine description contains the macro definitions for the AMOPs; these macro definitions expand into the object-language statements needed to interpret the corresponding abstract machine instructions. A macro definition consists of a list of AMOP labels followed by a list of character string constants. The list of AMOP labels specify that abstract machine instructions for these AMOPs are to be emitted as macro calls which refer to this macro definition. The character strings make up the body of the macro definition; they are written out in sequence as the expansion of a corresponding macro call. The character strings may have optional location prefixes which test for a specific set of locations of the operands and result; a character string with an attached location prefix is included in the expansion of the macro call only if the test specified by the location prefix succeeds. A character string may contain embedded macro calls and references to the arguments of the macro call (see Appendix VI, section 4). The macro definition for an AMOP must correspond to the location definition for the AMOP in that correct code must be generated for all combinations of operand/result locations that are allowed by the location definition.

The macro definitions can refer to the AMOP and the operand/result locations by using the following abbreviations:

abbreviation	expansion	meaning
*0	%n(*0)	symbolic representation of operation
*F	%n(*3,*4)	symbolic representation of first operand
*S	%n(*5,*6)	symbolic representation of second operand
*R	%n(*1,*2)	symbolic representation of result
*'O	*0	internal representation of operation
*'F	*3,*4	internal representation of first operand
*'S	*5,*6	internal representation of second operand
*'R	*1,*2	internal representation of result

Recall that in the intermediate language representation of an abstract machine instruction, the first argument of the macro call is the AMOP opcode, and the following arguments are REFs for the result, first operand, and second operand (see section 2.1.1.2). The macro "n" is the implementer-defined NAME macro which can return any convenient symbolic representation for an operation or operand/result location; it is assumed to be implemented as a C routine called ANAME (see Appendix VI, section 4).

An example of a simple macro definition is the definition for integer addition used in the HIS-6000 machine description. The location definition is

```
+i:             r,M,l;
```

and the macro definition is

```
+i:          "      AD*R    *S"
```

This location/macro definition of the AMOP '+i' expands to produce assembly-language statements such as

```
ADA     X       (external variable "X")
ADQ     3,DL    (literal "3")
ADA     0,2     (indirect through X2)
ADQ     5,7     (an automatic or temporary)
```

A more complicated macro definition is used for the AMOP '.ii' (move integer). This macro definition must be capable of generating code to move an integer between a memory location and a general register or from one general register to the other. Three character strings with location prefixes are used for the three cases register-to-memory, memory-to-register, and register-to-register:

```
.ii:
(r,,M):      "      ST*F    *R"
(M,,r):      "      LD*R    *F"
(r,,r):      "      LLR     36"
```

The location prefixes consist of location expressions for the first operand, second operand, and result. The operand and result locations of a particular macro call are compared to the location expressions in the location prefix (comparisons with a null location expression always succeed); if all three comparisons succeed, the corresponding character string is included in the expansion of the macro call.

The macro section of the machine description may also define explicitly named macros; these may be keyword macros (see section 2.1.2) or implementer-defined macros which are called in the definitions of other macros. A named macro is defined by using the name of the macro in place of an AMOP in the label(s) preceding the body of the macro defintion. A single macro definition may have both AMOP and macro name labels; this is useful when it is desired that the definition of one abstract machine instruction itself contain another abstract machine instruction since the "internal" names used to refer to the macro definitions of AMOPs are not accessible to the writer of the machine description. An example of a keyword macro definition in the HIS-6000 machine description is that for the ENTRY macro:

```
en:          "      SYMREF  *0"
```

The argument to the ENTRY macro is an assembler symbol as produced by the IDN macro (see Appendix III).

The macro section of the machine description consists of the reserved word "macros" followed by a sequence of macro definitions. Macro definitions must be provided for most of the AMOPs of the intermediate language (exceptions are indicated in Appendix II) and for all of the keyword macros of the intermediate language which are not defined by C routines. An AMOP or a macro name may not be defined more than once in the macro section of the machine description.

Appendix II - The Intermediate Language: AMOPs

The operations of the abstract machine are represented in the intermediate language as three-address instructions; the operators of these instructions, called abstract machine operators (AMOPs), are described in the tables below. For each AMOP is listed its opcode (in octal), its symbolic representation in the machine description, the types of its operands and result, and a description of the basic operation involved. The type entry consists of a list of types for the first operand, second operand (if any), and result of an AMOP, in that order; the types are taken from the following list of abbreviations:

c	character
i	integer
f	floating-point
d	double-precision floating-point
x	any type
p	any pointer
p0	class 0 pointer
p1	class 1 pointer
p2	class 2 pointer
p3	class 3 pointer
l	a location (the result of a jump)

The following notes are referenced in the AMOP tables:

1 - This AMOP should be defined only if the corresponding pointer classes are defined.
2 - The definition of this AMOP is optional.
3 - OPLOCs should not be specified for this AMOP.
4 - This AMOP is used only in the tree representation of expressions internal to the code generation phase: it should not appear in the machine description.
5 - This AMOP causes a side-effect on its (first) operand, which must be an lvalue; therefore, all OPLOCs for this AMOP must specify memory as the location of the (first) operand.

Unary Abstract Machine Operators

opcode	symbol	types	notes	basic operation
0000	-ui	i,i		unary minus
0001	-ud	d,d		unary minus
0002	++bi	i,i	5	pre-increment
0003	++ai	i,i	5	post-increment
0004	--bi	i,i	5	pre-decrement
0005	--ai	i,i	5	post-decrement
0006	.BNOT	i,i		bitwise negation
0007	!	x,i	4	truth-value negation
0012	.sw	i,i		switch
0013	++bc	c,i	5	pre-increment
0014	++ac	c,i	5	post-increment
0015	--bc	c,i	5	pre-decrement
0016	--ac	c,i	5	post-decrement
0017	&u0	x,p0		address of
0020	&u1	x,p1	1	address of
0021	&u2	x,p2	1	address of
0022	&u3	x,p3	1	address of
0023	*u	p,x	4	indirection
0024	==0p0	p0,l	2	jump on null pointer
0025	==0p1	p1,l	1,2	jump on null pointer
0026	==0p2	p2,l	1,2	jump on null pointer
0027	==0p3	p3,l	1,2	jump on null pointer
0030	!=0p0	p0,l	2	jump on non-null pointer
0031	!=0p1	p1,l	1,2	jump on non-null pointer
0032	!=0p2	p2,l	1,2	jump on non-null pointer
0033	!=0p3	p3,l	1,2	jump on non-null pointer

Conversion Abstract Machine Operators

opcode	symbol	types	notes	basic operation
0040	.ci	c,i		convert c to i
0041	.cf	c,f		convert c to f
0042	.cd	c,d		convert c to d
0043	.ic	i,c		convert i to c
0044	.if	i,f		convert i to f
0045	.id	i,d		convert i to d
0046	.ip0	i,p0		convert i to p0
0047	.ip1	i,p1	1	convert i to p1
0050	.ip2	i,p2	1	convert i to p2
0051	.ip3	i,p3	1	convert i to p3
0052	.fc	f,c		convert f to c
0053	.fi	f,i		convert f to i
0054	.fd	f,d		convert f to d
0055	.dc	d,c		convert d to c
0056	.di	d,i		convert d to i
0057	.df	d,f		convert d to f
0060	.p0i	p0,i		convert p0 to i
0061	.p0p1	p0,p1	1	convert p0 to p1
0062	.p0p2	p0,p2	1	convert p0 to p2
0063	.p0p3	p0,p3	1	convert p0 to p3
0064	.p1i	p1,i	1	convert p1 to i
0065	.p1p0	p1,p0	1	convert p1 to p0
0066	.p1p2	p1,p2	1	convert p1 to p2
0067	.p1p3	p1,p3	1	convert p1 to p3
0070	.p2i	p2,i	1	convert p2 to i
0071	.p2p0	p2,p0	1	convert p2 to p0
0072	.p2p1	p2,p1	1	convert p2 to p1
0073	.p2p3	p2,p3	1	convert p2 to p3
0074	.p3i	p3,i	1	convert p3 to i
0075	.p3p0	p3,p0	1	convert p3 to p0
0076	.p3p1	p3,p1	1	convert p3 to p1
0077	.p3p2	p3,p2	1	convert p3 to p2

Binary Abstract Machine Operators

opcode	symbol	types	notes	basic operation
0100	+i	i,i,i		addition
0101	=+i	i,i,i	2,5	addition-assignment
0102	+d	d,d,d		addition
0103	=+d	d,d,d	2,5	addition-assignment
0104	-i	i,i,i		subtraction
0105	=-i	i,i,i	2,5	subtraction-assignment
0106	-d	d,d,d		subtraction
0107	=-d	d,d,d	2,5	subtraction-assignment
0110	*i	i,i,i		multiplication
0111	=*i	i,i,i	2,5	multiplication-assignment
0112	*d	d,d,d		multiplication
0113	=*d	d,d,d	2,5	multiplication-assignment
0114	/i	i,i,i		division
0115	=/i	i,i,i	2,5	division-assignment
0116	/d	d,d,d		division
0117	=/d	d,d,d	2,5	division-assignment
0120	%	i,i,i		modulo
0121	=%	i,i,i	2,5	modulo-assignment
0122	<<	i,i,i		left-shift
0123	=<<	i,i,i	2,5	left-shift-assignment
0124	>>	i,i,i		right-shift
0125	=>>	i,i,i	2,5	right-shift-assignment
0126	&	i,i,i		bitwise AND
0127	=&	i,i,i	2,5	bitwise AND-assignment
0130	^	i,i,i		bitwise XOR
0131	=^	i,i,i	2,5	bitwise XOR-assignment
0132	.OR	i,i,i		bitwise OR
0133	=OR	i,i,i	2,5	bitwise OR-assignment
0134	&&	x,x,i	4	truth-value AND
0135	.TVOR	x,x,i	4	truth-value OR
0136	-p0p0	p0,p0,i		pointer subtraction
0137	=	x,x,x	4	assignment
0146	+p0	p0,i,p0		increment pointer by
0147	+p1	p1,i,p1	1	increment pointer by
0150	+p2	p2,i,p2	1	increment pointer by
0151	+p3	p3,i,p3	1	increment pointer by
0152	-p0	p0,i,p0		decrement pointer by
0153	-p1	p1,i,p1	1	decrement pointer by
0154	-p2	p2,i,p2	1	decrement pointer by
0155	-p3	p3,i,p3	1	decrement pointer by

Abstract Machine Operators, continued

opcode	symbol	types	notes	basic operation
0160	.cc	c,c	3	move character
0161	.ii	i,i	3	move integer
0162	.ff	f,f	3	move float
0163	.dd	d,d	3	move double
0164	.p0p0	p0,p0	3	move pointer p0
0165	.p1p1	p1,p1	1,3	move pointer p1
0166	.p2p2	p2,p2	1,3	move pointer p2
0167	.p3p3	p3,p3	1,3	move pointer p3
0171	?	x,x,x	4	conditional
0172	:	x,x,x	4	conditional
0200	==i	i,i,l		jump on equal
0201	!=i	i,i,l		jump on not equal
0202	<i	i,i,l		jump on less than
0203	>i	i,i,l		jump on greater than
0204	<=i	i,i,l		jump on less than or equal
0205	>=i	i,i,l		jump on greater than or equal
0206	==d	d,d,l		
0207	!=d	d,d,l		
0210	<d	d,d,l		
0211	>d	d,d,l		
0212	<=d	d,d,l		
0213	>=d	d,d,l		
0214	==p0	p0,p0,l		
0215	!=p0	p0,p0,l		
0216	<p0	p0,p0,l		
0217	>p0	p0,p0,l		
0220	<=p0	p0,p0,l		
0221	>=p0	p0,p0,l		
0222	==p1	p1,p1,l	1,2	
0223	!=p1	p1,p1,l	1,2	
0224	<p1	p1,p1,l	1,2	
0225	>p1	p1,p1,l	1,2	
0226	<=p1	p1,p1,l	1,2	
0227	>=p1	p1,p1,l	1,2	
0230	==p2	p2,p2,l	1,2	
0231	!=p2	p2,p2,l	1,2	
0232	<p2	p2,p2,l	1,2	
0233	>p2	p2,p2,l	1,2	
0234	<=p2	p2,p2,l	1,2	
0235	>=p2	p2,p2,l	1,2	
0236	==p3	p3,p3,l	1,2	
0237	!=p3	p3,p3,l	1,2	
0240	<p3	p3,p3,l	1,2	
0241	>p3	p3,p3,l	1,2	
0242	<=p3	p3,p3,l	1,2	
0243	>=p3	p3,p3,l	1,2	

Abstract Machine Operators, continued

opcode	symbol	types	notes	basic operation
0260	++bp0	p0,i,p0	5	pre-increment by
0261	++ap0	p0,i,p0	5	post-increment by
0262	--bp0	p0,i,p0	5	pre-decrement by
0263	--ap0	p0,i,p0	5	post-decrement by
0264	++bp1	p1,i,p1	1,5	pre-increment by
0265	++ap1	p1,i,p1	1,5	post-increment by
0266	--bp1	p1,i,p1	1,5	pre-decrement by
0267	--ap1	p1,i,p1	1,5	post-decrement by
0270	++bp2	p2,i,p2	1,5	pre-increment by
0271	++ap2	p2,i,p2	1,5	post-increment by
0272	--bp2	p2,i,p2	1,5	pre-decrement by
0273	--ap2	p2,i,p2	1,5	post-decrement by
0274	++bp3	p3,i,p3	1,5	pre-increment by
0275	++ap3	p3,i,p3	1,5	post-increment by
0276	--bp3	p3,i,p3	1,5	pre-decrement by
0277	--ap3	p3,i,p3	1,5	post-decrement by

Appendix III - The Intermediate Language: Keyword Macros

The keyword macros of the intermediate language are described below in alphabetical order. Each section is headed by the name of a macro and its calling sequence; following is a description of the arguments and the intended function of the macro call.

1. ADCONn: zAn(NAME) [n=0,1,2,3]

This is a set of macros, one for each possible pointer class. NAME is an object-language symbol constructed from an identifier by the IDN macro. The expansion of an ADCONn macro should define a pointer constant of pointer class "n" which points to the external variable or function with the given name. This macro is used in the initialization of static and external pointers and arrays of pointers.

2. ALIGN: zAL(N)

N is an integer specifying the CTYPE (an internal type specification) of an object for which the appropriate alignment of the location counter must be made. The relevant CTYPEs are:

value	ctype
2	char
3	int
4	float
5	double
6-9	pointer

The expansion of the macro call should be the pseudo-operations needed (if any) to properly align the location counter. This macro is used in the initialization of static and external variables.

3. CALL: zCA(NARGS,ARGP,0,FBASE,FOFFSET)

The CALL macro generates a function call. NARGS is an integer specifying the number of arguments to the function call; ARGP is an integer specifying the byte offset in the caller's stack frame of the arguments which have been so placed by previous instructions. FBASE and FOFFSET are integers which together make up a REF specifying the location of the function being called (it may be indirect through a pointer in a register); these are passed as arguments 3 and 4 of the macro call so that they may be referenced as *F in the macro definition.

4. CHAR: zC(I)

The CHAR macro produces a definition of a character constant whose value is the integer I; it is used in the initialization of static and external characters and arrays of characters. When producing code for an assembler which does not have a byte location counter (for example, the HIS-6000 assembler GMAP), the characters produced by CHAR macro calls must be stored in a buffer until either enough are accumulated to fill a machine word or a macro call other than CHAR is issued; in this case, all macros which may follow a CHAR macro must first check to see if there are any characters in the buffer and if so, print the appropriate statement and clear the buffer.

5. DOUBLE: zD(I)

The DOUBLE macro produces a definition of a non-negative double-precision floating-point constant whose C source representation is stored in the internal compiler table CSTORE at an offset specified by the integer I (the compiler itself does not use any floating-point operations). This macro is used in the initialization of static and external double-precision floating-point variables and arrays.

6. END: zEND()

The END macro marks the end of the intermediate language program. It may produce an END statement, if needed, or signal that any processing associated with the end of the program should be performed.

7. ENTRY: zEN(NAME)

NAME is an object language symbol constructed from an identifier by the IDN macro. The expansion of the ENTRY macro should define the symbol as an entry point, that is, one which is defined in the current program but accessible to other programs.

8. EPILOG: zEP(FUNCNO,FRAMESIZE)

The EPILOG macro produces the epilog code for a C function. The epilog code should restore the environment of the calling function and return to that function. In the HIS-6000 implementation, these actions are performed by a subroutine. FUNCNO and FRAMESIZE are integers which specify the internal function number of the function and the size in bytes of its stack frame, respectively. In the HIS-6000 implementation, these integers are used to define an assembly-language symbol whose value is the size in words of the stack frame; this symbol is used by the code produced by the PROLOG macro which allocates the stack frame.

9. EQU: zEQ(NAME)

NAME is an object language symbol constructed from an identifier by the IDN macro; it is to be defined as having a value equal to the current value of the location counter.

10. EXTRN: zEX(NAME)

The EXTRN macro is similar to the ENTRY macro except that it defines the symbol to be an external reference, that is, one which is used in the current program but assumed to be defined in another program.

11. FLOAT: zF(I)

The FLOAT macro produces a definition of a non-negative single-precision floating-point constant; the argument has the same interpretation as that of the DOUBLE macro.

12. GOTO: zGO(0,BASE,OFFSET)

The GOTO macro produces an unconditional jump to a location denoted in the source program by a label constant or expression. BASE and OFFSET together make up a REF which specifies the target location of the jump; these are passed as arguments 1 and 2 of the macro call so that they may be referenced as #R in the macro definition.

13. HEAD: zHD()

The HEAD macro marks the beginning of the intermediate language program. It may produce header statements, if needed, or signal that any initialization processing should be performed.

14. IDN: zI(X)

The IDN macro should expand to the object language representation of the identifier whose C source representation is stored in the internal compiler table CSTORE at an offset specified by the integer X. The processing performed by this macro may include the truncation of long names, the replacement of the underline character (which is allowed in C identifiers), and the insertion of special character(s) to avoid conflicts between C identifiers and other object language symbols.

15. INT: zIN(I)

The INT macro produces a definition of an integer constant whose value is specified by the integer I. It is used in the initialization of static and external variables and arrays and in the construction of tables for the LSWITCH macro.

16. LABCON: zLC(N)

The LABCON macro generates an address constant whose value is the address corresponding to internal label number N. The LABCON macro is used to construct the tables for the LSWITCH and TSWITCH macros.

17. LABDEF: zL(N)

The LABDEF macro defines the location of internal label number N.

18. LN: zLN(N)

The LN macro associates the line in the source program whose line number is specified by the integer N with the current value of the location counter. This macro may optionally produce a comment line in the object program to aid in the reading of the object program, or it may define a line-number symbol to be used in conjunction with a debugging system.

19. LSWITCH: zLS(N,LBASE,LOFFSET,IBASE,IOFFSET)

The LSWITCH macro should generate code which jumps according to the value of the integer whose location is given by IBASE and IOFFSET (selected from the locations permitted by the OPLOC for the .sw operation). This macro is immediately followed by N (N>0) INT macros (the cases), which are immediately followed by N LABCON macros (the corresponding labels). A search should be made through the case list; if a match is found, a jump should be made to the label defined by the corresponding LABCON macro. If the integer matches none of the list entries, then a jump should be made to the internal label defined by LBASE and LOFFSET.

20. NDOUBLE: zND(I)

The NDOUBLE macro is the same as the DOUBLE macro except that the value of the defined constant is made negative.

21. NFLOAT: zNF(I)

The NFLOAT macro is the same as the FLOAT macro except that the value of the defined constant is made negative.

22. PROLOG: zP(FUNCNO,FUNCNAME)

The PROLOG macro produces the prolog code for a C function. FUNCNAME is an integer representing the name of the function as it appears in the source program; its interpretation is the same as that of the argument of the IDN macro. FUNCNO is an integer which specifies the internal function number of the function; it may be used in conjunction with the EPILOG macro to access the size of the function's stack frame. The PROLOG macro should define the entry point name and produce the code necessary to save the environment of the calling function and to set up the environment of the called function using the information provided in the function call. In the HIS-6000 implementation, these actions are performed by a subroutine. The PROLOG macro call appears in the intermediate language program immediately before the first instruction of the corresponding function.

23. RETURN: zRT()

The RETURN macro produces the statements needed to return from a function to the calling function; in general, this macro will result in a transfer to the EPILOG code. The returned value of the function is loaded by preceding macro calls into the appropriate register as specified in the RETURNREG statement of the machine description.

24. STATIC: zST(N,S)

The STATIC macro defines the location of the static variable whose internal static variable number is N. S is the size of the static variable in bytes. Typically, this macro will define an assembly language symbol by which the static variable can be referenced.

25. STRCON: zSC(N)

The STRCON macro should generate a character pointer which points to the string constant whose internal string number is N. The STRCON macro is used in the initialization of static and external variables.

26. STRING: zSR()

The STRING macro marks the place in the object program where the string constants should be defined. This macro is implemented as a C routine macro since substantial processing is involved.

27. TSWITCH: zTS(LO,LBASE,LOFFSET,IBASE,IOFFSET,HI)

The TSWITCH macro produces an indexed jump based on the value of the integer whose location is given by IBASE and IOFFSET (selected from the locations permitted by the OPLOC for the .sw operation). This macro is immediately followed by a sequence of HI-LO+1 LABCON macros defining the target labels corresponding to integer values from LO to HI. Values outside this range should result in transfers to the internal label defined by LBASE and LOFFSET.

28. ZERO: zZ(I)

The ZERO macro specifies the definition of a block of storage initialized to zero; the size in bytes of this storage area is specified by the integer I.

Appendix IV - The HIS-6000 Machine Description

The machine description used in the HIS-6000 implementation is listed below. Much of its complexity is a direct result of the fact that the HIS-6000 is not byte-addressed. In the macro definitions, the character sequence '\n' represents the newline character.

```
typenames (char,int,float,double);
regnames (x0,x1,x2,x3,x4,a,q,f);
memnames (reg,auto,ext,stat,param,label,intlit,floatlit,stringlit,ix0,ix1,ix2,ix3,ix4,ia,iq);
size 1(char),4(int,float),8(double);
align 1(char),4(int,float),8(double);
class x(x0,x1,x2,x3,x4), r(a,q);
conflict (a,f),(q,f);
saveareasize 16;
pointer p0(1), p1(4);
returnreg q(int,p0,p1),f(double);
type char(r),int(r),float(f),double(f),p0(r),p1(x);

.sw:                                        a,,1[x4];
+p0: -p0: +i:-i:&:^: .OR: -p0p0: <<: >>:    r,M,1;
+p1:                                        M,M,x;
-p1:                                        x,q,1;
=+i: =&: =^: =OR:                           M,r,1;
*i: /i:                                     q,M,q[a];
+d: -d: *d: /d:                             f,M,f;
%:                                          q,M,a[q];
=<<: =>>:                                   M,a,q; M,q,a;
&u:                                         M,,x;
                                  auto|ext|stat|stringlit|ia|iq,,r;
.BNOT: .ic: .ci:                            r,,1;
-ui: --bi:                                  M,,r;
.cf: .cd: .if: .id:                         a,,f;
.fc: .dc: .fi: .di:                         f,,q;
.fd:                                        M,,f;
.df: -ud:                                   f,,1;
.ip0: .p0i:                                 r,,1;
                                            M,,r;
.ip1: .p0p1:                                r,,x;
                                            M,,x;
.p1i: .p1p0:                                x,,r;
                                            M,,r;
++bi:                                       M,,1;
++ai: --ai: ++bc: ++ac:
--bc: --ac:                                 M,,a[q];
                                            M,,q[a];
++bp: --bp:                                 M,M,r|x;
++ap: --ap:                                 M,M,a[q];
                                            M,M,q[a];
                                            M,M,x;
==0: !=0: <0: >0: <=0: >=0:                 r|f,,r;
==p: !=p: <p: >p: <=p: >=p:                 r|x,M,M;
```

macros

```
.sw: "              TSX5    .SWTCH"

.ci:                "\\"

.cc:
(auto„): "          EA*R    0,7\n"
(stat„): "          EA*R    .STAT\n"
(ia„q): "           STA     .TEMP
                    LDQ     .TEMP\n"
(iq„a): "           STQ     .TEMP
                    LDA     .TEMP\n"
(auto|stat|indirect„):
"%if(%o(*'F),       AD*R    %co(*'F)\n,)
                    TSX5    .CTO*R"
(ext|stringlit„):
"                   LD*R    *F
                    *RRL    27"
(r„r): "            EA*R    0,*FL
                    *RRL    18"
(r„auto|stat|indirect|stringlit):
"                   EAX5    0,*FL\n"
(r„auto): "         EA*F    0,7\n"
(r„stat): "         EA*F    .STAT\n"
(r„auto|stat): "%if(%o(*'R),    AD*F    %co(*'R)\n,)
                    TSX4    .*FTOC"
(r„stringlit): "    EA*F    *R
                    TSX4    .*FTOC"
(r„ext): "          *FLS    27
                    ST*F    *R
                    *FRL    27"
(q„ia):
"%if(%o(*'R),       ADA     %co(*'R)\n,)
                    TSX4    .ATOC"
(a„iq):
"%if(%o(*'R),       ADQ     %co(*'R)\n,)
                    TSX4    .QTOC"

.ii:
(r„M): "            ST*F    *R"
(M„r): "            LD*R    *F"
(r„r): "            LLR     36"

.ff:
(f„M): "            FSTR    *R"
(M„f): "            FLD     *F"

.dd:
(f„M): "            DFST    *R"
(M„f): "            DFLD    *F"

.p0p0:
(r„r): "            LLR     36"
(r„M): "            ST*F    *R"
(M„r): "            LD*R    *F"
```

```
.p1p1:
(x„x): "        EA*R    0,*3"
(x„M): "        STZ     *R
                ST*F    *R"
(M„x): "        LD*R    *F"
(x„q): "        EAQ     0,*3"
(q„x): "        EA*R    0,QU"
(M„q): "        LDQ     *F"
(q„M): "        STQ     *R"

.p0p1:
(r„x): "        EA*R    0,*FU"
(M„x): "        LD*R    *F"

.p1p0:
(x„r): "        EA*R    0,*3"
(M„r): "        LD*R    *F"

.ic: "          AN*F    =0377,DL"

.ip0:
(M„r): "        LD*R    *F"
(r„r):          "\\"

.ip1:
(r„x): "        EA*R    0,*FU"

.p0i:
(M„r): "        LD*R    *F"
(r„r):          "\\"

.p1i:
(x„r): "        EA*R    0,*3"

.fd: "          FLD     *F"

.df:            "\\"

.cf: .cd: .if: .id: "   LDQ     0,DL
                LDE     =35B25,DU
                FNO"

.fi: .di: "     UFA     =71B25,DU"

.fc: .dc: "     UFA     =71B25,DU
                ANQ     =0377,DL"

+i: "           AD*R    *S"
-i: "           SB*R    *S"
*i: "           MPY     *S"
/i: %: "        DIV     *S"
+d: "           DFAD    *S"
-d: "           DFSB    *S"
*d: "           DFMP    *S"
/d: "           DFDV    *S"
=+i: "          AS*S    *R"
```

```
>>:
(,intlit,): "       *FRS    %o(*'S)"
(,~intlit,): "      LXL5    *$
                    *FRS    0,5"

<<:
(,intlit,): "       *FLS    %o(*'S)"
(,~intlit,): "      LXL5    *$
                    *FLS    0,5"

=>>: "              LD*R    *F
                    *RRS    0,*SL
                    ST*R    *F"

=<<: "              LD*R    *F
                    *RLS    0,*SL
                    ST*R    *F"

+p0: "              *FRS    16
                    AD*F    *$
                    *FLS    16"

-p0: "              *FRS    16
                    SB*F    *$
                    *FLS    16"

+p1: "              LXL*1   *$
                    ADL*R   *F"

-p1: "              QLS     18
                    STQ     .TEMP
                    SBL*F   .TEMP"

-ui: "              LC*R    *F"

--bi: "             LD*R    *F
                    SB*R    =1,DL
                    ST*R    *F"

-ud: "              FNEG"

++bi: "             AOS     *F"

++ai: "             LD*R    *F
                    AOS     *F"

--ai: "             LDA     *F
                    LDQ     *F\n"
(,,a): "            SBQ     =1,DL
                    STQ     *F"
(,,q): "            SBA     =1,DL
                    STA     *F"
```

```
++bp:
(„x): "          LD*R     *F
                 EA*R     %o(*'S)/4,*1
                 ST*R     *F"
(„r): "          LD*R     *F
                 ADL*R    %co(*'S)
                 ST*R     *F"

--bp:
(„x): "          LD*R     *F
                 EA*R     -%o(*'S)/4,*1
                 ST*R     *F"
(„r): "          LD*R     *F
                 SBL*R    %co(*'S)
                 ST*R     *F"

++ap:
(„x): "          LD*R     *F
                 EAX5     %o(*'S)/4,*1
                 STX5     *F"
(„a|q): "        LDA      *F
                 LDQ      *F\n"
(„a): "          ADLQ     %co(*'S)
                 STQ      *F"
(„q): "          ADLA     %co(*'S)
                 STA      *F"

--ap:
(„x): "          LD*R     *F
                 EAX5     -%o(*'S)/4,*1
                 STX5     *F"
(„a|q): "        LDA      *F
                 LDQ      *F\n"
(„a): "          SBLQ     %co(*'S)
                 STQ      *F"
(„q): "          SBLA     %co(*'S)
                 STA      *F"

.BNOT: "         ER*F     *-1"

&u:
(ia|iq„r):
"%if(%o(*'F),    ADL*F    %co(*'F)\n,)\\"
(ia„q): "        LLR      36"
(iq„a): "        LLR      36"
(auto|stat„r): " EA*R     %n(*3,0)
%if(%o(*'F),     ADL*R    %co(*'F)\n,)\\"
(ext|stringlit„r): " EA*R *F"
(„x): "          EA*R     *F"

&: "             AN*F     *S"
=&: "            ANS*S    *F"
^: "             ER*F     *S"
=^: "            ERS*S    *F"
.OR: "           OR*F     *S"
=OR: "           ORS*S    *F"
```

```
==p: "          CMP*F   *S
                TZE     *R"
!=p: "          CMP*F   *S
                TNZ     *R"
<p: "           CMP*F   *S
                TZE     *+2
                TNC     *R"
>p: "           CMP*F   *S
                TZE     *+2
                TRC     *R"
<=p: "          CMP*F   *S
                TZE     *R
                TNC     *R"
>=p: "          CMP*F   *S
                TRC     *R"

jc:
(„f): "         DFCMP   =0D0\n"
(„r): "         CMP*R   0,DL\n"
                "%ajc(*0,*2)"

¬p0p0: "        SBL*F   *S
                *FRL    16"

hd: "$          GMAP"

jmp: "          TRA     *0"

o:              "*1"

en: "           SYMDEF  *0"

ex: "           SYMREF  *0"

st: "           SYMREF  .PROLG,EPILG,TEMP,SWTCH
                SYMREF  .CTOA,CTOQ,ATOC,QTOC
.STAT           EQU     *"

p: "%idn(*1)    EQU     *
                TSX0    .PROLG
                ZERO    .FS*0"

co:             "=V20/*1,16/0"

ce: "           TSX1    *F
                ZERO    *1/4,*0"

rt: "           TRA     .EPILG"

ep: "           TRA     .EPILG
.FS*0           EQU     *1/4"

go: "           TRA     *R"
```

```
cpq:
(auto„): "          EAQ     0,7\n"
(stat„): "          EAQ     .STAT\n"
(ia„): "            LLR     36\n"
(auto|stat|indirect„):
"%if(%o(#'F),       ADQ     %co(#'F)\n,)\\"

++bc:
(auto|stat|indirect„):
"%cpq(0,0,0,#'F)
                    STQ     .TEMP
                    LDA     .TEMP
                    TSX5    .CTOA
                    ADA     1,DL
                    ANA     =0377,DL
                    EAX5    0,AL
                    TSX4    .QTOC"
(ext„): "           LDA     #F
                    ADA     =01000,DU
                    STA     #F"

--bc:
(auto|stat|indirect„):
"%cpq(0,0,0,#'F)
                    STQ     .TEMP
                    LDA     .TEMP
                    TSX5    .CTOA
                    SBA     1,DL
                    ANA     =0377,DL
                    EAX5    0,AL
                    TSX4    .QTOC"
(ext„): "           LDA     #F
                    SBA     =01000,DU
                    STA     #F"

++ac:
(auto|stat|indirect„):
"%cpq(0,0,0,#'F)
                    STQ     .TEMP
                    LDA     .TEMP
                    TSX5    .CTOA
                    EAX5    1,AL
                    TSX4    .QTOC"
(ext„): "           LDA     #F
                    LDQ     #F
                    ADQ     =01000,DU
                    STQ     #F"
```

```
--ac:
(auto|stat|indirect_n):
"%cpq(0,0,0,*F)
                STQ     .TEMP
                LDA     .TEMP
                TSX5    .CTOA
                EAX5    -1,AL
                TSX4    .QTOC"
(ext_n): "      LDA     *F
                LDQ     *F
                SBQ     =01000,DU
                STQ     *F"
```

(The set of AMOPs used in the machine description is slightly different than the set presented in Appendix II, particularly with regard to the unary [illegible] increment/decrement [illegible] corrected after the [illegible]

Appendix V - The HIS-6000 C Routine Macro Definitions

The C routine macro definitions used in the HIS-6000 implementation are listed on the following pages. A C routine macro definition is written as a C function returning a character string value. This character string is "substituted" for the macro call and rescanned by the macro expander; thus, it may contain references to its arguments and embedded macro calls. The formal parameters of the C routine are ARGC and ARGV: ARGC is an integer specifying the number of (character-string) arguments present in the associated macro call; ARGV is an array of pointers to those arguments.

When the following routines were written, the formatted print routine PRINT was capable of producing output only onto a file and not into a string in core; thus, where formatting is necessary, these routines print their output directly and return the null string. Although there are dangers inherent in this practice, in these cases the effect is the same as if the formatted string were returned and printed normally. The character sequences '\t', '\n', and '\\' represent tab, newline, and backslash, respectively.

```
char *fn[]
{"in","c","f","nf","d","nd","al","ajc",
"ad","z","i","sr","end","n","eq","ln",
"other","if"};
char (*ff[])()
        {aint,achar,afloat,anegf,adouble,anegd,aalign,ajc,
        aadcon,azero,aidn,astring,aend,aname,aequ,aln,
        other,aif};
int     nfn 18,
        lineno 0,
        mflag 0,
        packb[4],
        packno;

char *aln(argc,argv) int argc; char *argv[];

{lineno=atoi(argv[0]);
packf();
return(".N=0         EQU     *");
}

char *aequ(argc,argv) int argc; char *argv[];

{packf();
return("*OEQU       *");
}

char *aint(argc,argv) int argc; char *argv[];

{packf();
return("\tDEC\t=0");
}

char *achar(argc,argv) int argc; char *argv[];

{if (argc>0) packc(atoi(argv[0]));
return("\\");          /* conceal following newline */
}

char *afloat(argc,argv) int argc; char *argv[];

{packf();
if (argc>0) print("\tDEC\t%m",atoi(argv[0]));
return("");
}

char *adouble(argc,argv) int argc; char *argv[];

{

packf();
if (argc>0)
        {print("\tDEC\t");
        return(adbic(atoi(argv[0])));
        }
return("");
```

```
}

char *anegf(argc,argv) int argc; char *argv[];

{packf();
if (argc>0) print("\tDEC\t-%m",atoi(argv[0]));
return("");
}

char *anegd(argc,argv) int argc; char *argv[];

{

packf();
if (argc>0)
        {print("\tDEC\t-");
        return(adbic(atoi(argv[0])));
        }
return("");
}


char *astring(argc,argv) int argc; char *argv[];

{auto int i,f,lc,c;
auto char *cp;

lc=0;       /* location counter in STRING file */
f=xopen(pname,fn_string,MREAD,BINARY);

while(1)
        {packf();
        c=cgetc(f);
        if(ceof(f)) break;
        print(".S%d\tEQU\t*\n",lc);
        lc++;
        while(1)
                {if (c=='$')
                        {c=cgetc(f);
                        lc++;
                        if (c=='0') c='\0';
                        packc(c);
                        }
                else
                        {packc(c);
                        if (!c) break;
                        }
                c=cgetc(f);
                lc++;
                }
        }
cclose(f);
return("\\");
}

char *aend(argc,argv) int argc; char *argv[];
```

```
{packf();
return("\tEND");
}

char *regnames[] {"X0","X1","X2","X3","X4","A","Q",""};

char *aname(argc,argv) int argc; char *argv[];

{auto int base,offset;

if (argc>1) offset=atoi(argv[1]); else offset=0;
if (argc>0) base=atoi(argv[0]); else base=0;
if (mflag) cprint("ANAME(%d,%d)\n",base,offset);
if (base>=0) return(regnames[base]);
base = -base;
if (base >= c_indirect)
        {print("%d,%d",offset/4,base-c_indirect);
        goto check;
        }
else switch(base) {

case c_auto:
        print("%d,7",offset/4);
        goto check;
case c_extdef:
        return("%i(*1)");
case c_static:
        print(".STAT+%d",offset/4);
        goto check;
case c_param:
        print("%d,6",offset/4);
        goto check;
case c_label:
        print(".L%d",offset);
        break;
case c_integer:
        if (offset<0 || offset>32000) print("=%d",offset);
        else print("%d,DL",offset);
        break;
case c_float:
        print("=%s",adbic(offset));
        break;
case c_string:
        print(".S%d",offset);
        break;
        }
return("");
check:
if (offset%4) error(6025,lineno);
return("");
}

/*************************************************************************

        AALIGN - align location counter
```

```
*/

char *aalign(argc,argv) int argc; char *argv[];

{

switch(atoi(argv[0])) {
case ct_double:
            packf();
            return("\tEVEN");
            }
return("\\");
}

/*************************************************************************

            AJC - emit conditional jump

*/

char *ajc(argc,argv) int argc; char *argv[];

{auto int cond;

cond=atoi(argv[0]);

switch(cond) {

case cc_eq0:        return("\tTZE\t*1");
case cc_ne0:        return("\tTNZ\t*1");
case cc_lt0:        return("\tTMI\t*1");
case cc_ge0:        return("\tTPL\t*1");
case cc_gt0:        return("\tTZE\t*+2\n\tTPL\t*1");
case cc_le0:        return("\tTZE\t*1\n\tTMI\t*1");
            }
return("");
}

char *other(argc,argv) int argc; char *argv[];

{switch(atoi(argv[0])) {
case 5:    return("Q");
case 6:    return("A");
            }
return("BAD");
}

char *aif(argc,argv) int argc; char *argv[];

{return(atoi(argv[0])?"*1":"*2");
}

/*          PACK CHARACTERS INTO WORDS          */

packc(i) int i;
```

```
{

packb[packno++]=i;
if (packno>=4)
        {print("\tVFD\t9/%d,9/%d,9/%d,9/%d\n",
                packb[0],packb[1],packb[2],packb[3]);
        packno=0;
        }
}

packf()

{

while(packno!=0) packc(0);
}


char *aadcon(argc,argv) int argc; char *argv[];

{packf();
return("\tZERO\t=0");
}

char *azero(argc,argv) int argc; char *argv[];

{auto int i,j;

if (argc>0)
        {i=atoi(argv[0]);
        while(packno && i) {packc(0);i--;}
        j = i/4; i =% 4;
        if (j>0) print("\tBSS\t%d\n",j);
        while(i--)packc(0);
        }
return("\\");
}

char  *aidn(argc,argv) int argc; char *argv[];

{auto char *cp1,*cp2;
static char n[7];
auto int i,c;

if (argc>0)
        {cp1 = &cstore[atoi(argv[0])];
        cp2 = n;
        for(i=0;i<6;i++)
                {c = *cp1++;
                if (c == '_') c = 'J';
                *cp2++ = c;
                }
        *cp2='\0';
        return(n);
        }
return("");
```

```
}

adbic(i)

{auto char *cp1,*cp2;
static char buf[30];
auto int c,flag;

flag=FALSE;
cp1 = &cstore[i];
cp2 = &buf[0];

while(c = *cp1++)
        {if (c == 'E')
                {flag=TRUE;
                c = 'D';
                }
        if (cp2 < &buf[27])
                *cp2++ = c;
        }
if (!flag)
        {*cp2++ = 'D';
        *cp2++ = '0';
        }
*cp2++ = '\0';
return(&buf[0]);
}
```

Appendix VI - Overall Description of the Compiler

The compiler consists of four major phases. First, the lexical analysis phase (C1) transforms the source program into a string of lexical tokens such as identifiers, constants, and operators. Second, the syntactic analysis phase (C2) parses the token string and produces a tree representation of each function (procedure) defined in the source program. Third, the code generation phase (C3) transforms the trees produced by the syntactic analysis phase into an intermediate language program consisting of a sequence of macro calls representing instructions of the particular abstract machine defined by the implementer. Finally, the macro expansion phase (C4) expands the macro calls, producing an object language program as the output of the compiler. In addition, there is an error message editor (C5) which is invoked last in order to format any error messages produced by the other phases. The phases of the compiler are invoked in sequence by the control program (CC). The control program communicates with the various phases by passing as arguments to an invoked phase a set of character strings representing file names and an option list; the invoked phase returns a completion code which indicates whether or not any serious or fatal errors occurred during the execution of that phase. The various phases communicate with each other using intermediate files.

The lexical and syntax analysis phases may be run sequentially as described above, or, where a system's program size restrictions permit, may be combined into a single phase, thus eliminating the use of an intermediate file. This option is implemented through the use of compile-time conditionals. The remainder of this chapter will assume that the two phases are separate.

1. The Lexical Analysis Phase

The lexical analyzer reads in the source program and breaks it into a string of tokens such as identifiers, constants, and operators. The lexical analyzer also interprets compile-time control lines which allow one to include source from other files and to define manifest constants. The lexical analyzer produces output onto three intermediate files: the TOKEN file, which contains the string of tokens, the CSTORE file, which contains the source representations of identifiers and floating-point constants, and the STRING file, which contains character string constants. The TOKEN file is passed to the syntax analysis phase; the CSTORE and STRING files are not used until macro expansion. In addition, the lexical analyzer may write error messages in an internal form onto the ERROR file. A token is represented by a pair of integers called the TYPE and the INDEX of the token. The syntax analyzer performs its analysis on the basis of the token TYPE; thus most operators have a distinct TYPE, and there are separate TYPEs for identifiers, integer constants, floating-point constants, and character string constants. The INDEX is used to distinguish particular identifiers or constants; for example, the INDEX of an identifier is the index of the source representation of the identifier in the array of characters written onto the CSTORE file.

The main routine of the lexical analyzer consists of a loop which calls a routine GETTOK to return the next token in the input stream and then writes the token onto the TOKEN file. This loop also contains code to interpret compile-time control lines. GETTOK obtains input characters from a routine LEXGET which contains the logic to switch the input between the primary source file and "included" files. Except when processing character string constants, GETTOK translates the input characters using a translation table. On GCOS, this translation maps lower case into upper case, tabs into blanks, and carriage returns into newlines. This table would be changed when moving the compiler to a system using other than the ASCII character set. GETTOK partitions the character set into the following character classes:

1. letters
2. digits
3. apostrophe (')
4. quotation mark (")
5. newline
6. blank
7. period (.)
8. the escape character (\)
9. invalid characters
10. characters which are unambiguously single-character operators (such as '{')
11. characters which may begin a multi-character operator (such as '<' which may begin '<=')

GETTOK uses the character class of the current input character to determine its actions in analyzing the input string.

2. The Syntax Analysis Phase

The syntax analyzer accepts as input the token string generated by the lexical analyzer and produces output onto three intermediate files for the code generation phase: a tree representation of each function defined in the source program is written onto the NODE file; a symbol table containing declarative information about identifiers is written onto the SYMTAB file; and information regarding specified initial values of variables is written onto the INIT file.

The main routine of the syntax analysis phase is a table-driven LALR(1) parser. The tables are generated by a parser-generator YACC, written by S. C. Johnson [18]. The input to YACC is a BNF-like description of the syntax of C, augmented by action routines which are to be invoked by the parser when particular reductions are made. YACC analyzes the grammar and produces a set of tables written in C which are then compiled into the syntax analysis phase.

The tables produced by YACC represent instructions to the parser to test the TYPE of the current input token, to shift the current input token onto the stack, to perform a reduction and call an action routine, or to report a syntax error. When a syntax error is discovered, the parser writes error messages onto the ERROR file which give the current state of the parse. It then attempts to recover from the error so that any additional syntax errors in the program can meaningfully be reported. The parser attempts a recovery by popping states from the stack and/or skipping input tokens in various combinations. A recovery attempt is considered successful if the next five input tokens are shifted without detecting a new syntax error. If a recovery attempt is successful, error messages are written which describe the recovery actions taken and parsing is continued. If a successful recovery cannot be made within a limited region of the input program, the parser ceases execution after writing an error message.

The following C program illustrates the compiler's response to a syntax error, in this case unmatched parentheses:

```
int c;
int f(file)
{if ((c=getc(file) != 0) return(-1);
return(0);
}
```

The first error message, listed below, gives the state of the parse when the syntax error was discovered, followed by a cursor symbol '_', followed by the next five input tokens. The next error message indicates that the parser was able to recover from the error by skipping the next two input tokens. The resulting program, although syntactically correct, is meaningless. Therefore, in order to avoid extraneous error

messages, the code generation phase and the macro expansion phase are not executed after syntax errors have been detected.

```
3: SYNTAX ERROR. PARSE SO FAR: <ext_def_list> <function_dcl>
<block_head> IF ( <e> _ RETURN ( - 1 )
3: SKIPPED:  RETURN (
```

The following program also contains a syntax error due to unmatched parentheses; however, since there are no more right parentheses in the statement following the point where the error is detected, the parser recovers from the error by deleting the unfinished IF clause.

```
int c;
int f(file)
{if ((c=getc(file) == 0) c = -1;
return(c);
}
```

```
3: SYNTAX ERROR. PARSE SO FAR: <ext_def_list> <function_dcl>
<block_head> IF ( <e> _ C = - 1 ;
3: DELETED:  IF ( <e>
```

The following program is an example of a syntax error from which the parser could not recover within its allowed limits; thus, after skipping input tokens up to this limit, the parser gives up.

```
int c;
int f(file)
{if ((c=getc(file) != 0) c = 1;
else c = 0;
return(c);
}
```

```
3: SYNTAX ERROR. PARSE SO FAR: <ext_def_list> <function_dcl>
<block_head> IF ( <e> _ C = 1 ; ELSE
3: SKIPPED:  C = 1 ;
4: I GIVE UP
```

3. The Code Generation Phase

The code generation phase performs the following operations: (1) allocates storage for (determines the run-time locations of) variables, (2) performs type checks on operands and inserts conversion operators where necessary, (3) translates the tree representation of expressions into a more descriptive form with AMOPs, (4) performs some machine-independent optimizations on expressions, (5) emits macro calls to define names which may be referenced by other programs (ENTRY symbols) and to declare names which are assumed to be defined in other programs (EXTRN symbols), (6) emits macro calls to define and initialize variables, (7) emits macro calls to execute the control statements of each function defined in the source program, and (8) emits macro calls to evaluate expressions.

The code generation phase reads the NODE, SYMTAB, and INIT files produced by the syntax analysis phase and writes an intermediate language program in the form of macro calls onto two intermediate files, the MAC file and the HMAC file. The HMAC file contains the macro calls defining ENTRY symbols and EXTRN symbols which are produced last by the code generation phase but which, in some systems, may be required to appear at the beginning of the assembly language program. The MAC file contains the remainder of the intermediate language program.

The main routine of the code generation phase consists of a call to a routine SALLOC, which allocates run-

time storage and emits macro calls to define and initialize variables, followed by a loop which reads in the tree representation of a single C function from the NODE file and generates code (macro calls) for that function, followed by a call to a routine SDEF which emits macro calls to define ENTRY and EXTRN symbols.

The generation of code for a C function begins with a call to a routine FHEAD with the name of the function as an argument. FHEAD emits a PROLOG macro call which defines the entry point and produces code to set up the proper run-time environment. FHEAD then allocates storage in the run-time stack frame for the automatic variables of the function; storage is allocated for automatic variables in order of decreasing alignment requirement so that no space is wasted in the stack frame. The stack frame is assumed to be aligned according to the strictest of the alignment requirements of the various C data types (usually that of double-precision floating-point). A save area of the size specified in the machine description is reserved at the beginning of the stack frame.

The call to FHEAD is followed by a call to the routine STMT to generate code for the compound statement which is the body of the C function. The generation of code for the body of a C function occurs on two levels, the statement level and the expression level. The generation of code for statements is handled by the routine STMT which takes one argument, a pointer to a subtree representing a C statement. STMT is actually a very short routine which makes recursive calls to itself for the branches of a STATEMENT_LIST node and calls a larger routine ASTMT if the specified node is an actual statement (as opposed to a statement list). The purpose of splitting code generation for statements into the two routines STMT and ASTMT is to minimize the amount of stack space used while recursively descending the statement tree.

Following the call to STMT to generate code for the body of the C function, the size of the stack frame is adjusted to be a multiple of the stack alignment and an EPILOG macro call is emitted. On the HIS-6000, the EPILOG macro defines an assembly-language symbol whose value is the stack frame size; this symbol is referred to by the code produced by the PROLOG macro which allocates the stack frame.

4. The Macro Expansion Phase

The macro expansion phase expands the macro calls on the HMAC and MAC intermediate files using the information on the CSTORE and STRING intermediate files and places the result of that expansion on the output file. The macro expander is not a general-purpose macro processor; in particular, there are no built-in macro calls for defining macros or for handling local or global variables. Furthermore, the total number of characters (after any macro expansion) in the argument list of a macro call is limited to 100. The maximum allowed depth of nested macro calls is 10.

The macro expander processes a stream of characters terminated by a NULL character. Within this stream of characters, the characters '%', '*', and '\' have special significance. The '%' character indicates the beginning of a macro call, which consists of the '%', followed by the name of the macro, followed by a (possibly null) list of character string arguments separated by commas and enclosed in parentheses. The '*' character is used within the body of a macro definition to refer to the arguments of the macro call; the character sequences '*0' through '*9' refer to arguments 0 through 9, respectively. The '\' character is an escape character. The special interpretation of a character such as '%', '*', ')' or ',' is inhibited when that character is preceded by a '\'. In addition, the character sequences '\t', '\n', '\r' are used to represent tab, newline, and carriage-return, respectively. A '\' character followed by a newline character results in both characters being ignored; thus a macro which expands to a backslash will swallow the newline which followed the macro call in the input file. (A macro call in the input file which expands to the null string will leave a blank line in the compiler output; this is generally a sign that the implementer has not completely specified the macro definition for an AMOP.) The backslash character itself is represented by '\\'.

The normal operation of the macro expander consists of copying characters directly from the input stream to the output stream. When a '%' is encountered, the name of the macro and the arguments of the macro call are evaluated and collected in a buffer; this evaluation may itself involve the processing of embedded

macro calls. The input stream is then switched to the body of the macro definition and normal processing is resumed. When a '*' is encountered, the argument number is read and the input stream is switched to the corresponding character string argument of the current macro call, which is stored in the associated buffer. Normal processing is then resumed. The input stream operates in a stack-like manner in that when the end of a macro definition or an argument string is reached, the input stream is restored to its previous state. When end of file is reached on the HMAC file, the input stream is switched to the MAC file; when end of file is reached on the MAC file, macro expansion is terminated.

There are three types of macros which are handled by the macro expander. First, there are the macros representing three-address abstract machine instructions, which are produced by the code generator while processing expressions. These macros are defined only in the machine description; the macro calls are of a special form which directly specifies the internal number of the corresponding macro definition, as assigned by GT. For example, the macro call %3 refers to macro definition number 3. Second, there are the keyword macros which are produced by the code generator while processing function definitions and statements. These macros may be defined either in the machine description or by C routines; the macro calls specify the macro names as given in Appendix III. Finally, there are the macros which are created by the implementer and used within other macro definitions. These macros may be defined either in the machine description or by C routines; the macro calls specify the macro name as defined by the implementer.

A macro which is defined in the machine description is specified as a list of one or more character string constants, possibly with associated location prefixes for conditional expansion. Such a macro definition is implemented as a list of pointers to the character string constants, along with associated integers representing the conditions specified in the location prefixes, if any. The lists are accessed through an array MACDEF, produced by GT, which is indexed by the internal macro definition numbers assigned by GT to each macro definition in the machine description. As mentioned above, a macro call representing a three-address abstract machine instruction directly specifies the macro definition number. Other macros defined in the machine description are represented in a table produced by GT which associates the macro names with the corresponding macro definition numbers.

Macros defined by C routines are represented in a table provided by the implementer which associates the macro names with the corresponding C functions. This table consists of an array FN of pointers to the character string macro names, an array FF of pointers to the corresponding C functions, and an integer NFN specifying the number of entries in the table. It would be more convenient for the implementer to specify the C macro definitions in the machine description and let GT construct NFN, FN, and FF; however, this was not done because of the lexical difficulties associated with including C source in the machine description.

The macro expander is implemented as two levels of get-character routines. The lower level routine, GETC1, returns the next character from the current input source which may be either the input file (HMAC or MAC intermediate file) or a character string in memory. If it is a character string, it may be part of a definition of a macro specified in the machine description, an argument of the current macro call, or the result returned by a C routine macro definition. The current state of the input stream is kept in a stack of structures called input control blocks (ICBs); GETC1 uses the top ICB on the stack to determine the source of the next character. The members of an ICB structure are listed below with their meanings:

F — a flag indicating the type of the current input source (the input file, a macro defined in the machine description, or a character string)

LOCP — if the current input source is a macro defined in the machine description, this is a pointer to the current position in the list containing the pointers to the character strings which make up the macro definition

CP — if the current input source is not the input file, this is a pointer to the next character in the current character string

ARGV[10] — an array of pointers to the character string arguments of the current macro call

BASE[3] — the REF.BASEs of the result, the first operand, and the second operand of the current macro call, used when computing conditional expansion

A NULL character indicates the end of a character string or end-of-file on an input file; thus if the current input character is NULL, GETC1 updates the current state of the input stream by advancing LOCP or by popping an ICB off the stack or by switching the input file from the HMAC to the MAC intermediate file. GETC1 returns the NULL character only upon end-of-file on the MAC intermediate file.

The higher level get-character routine is MGET, which implements the '*', '%', and '\' conventions. MGET begins by calling GETC1 to obtain a character. If the character returned is a backslash, then GETC1 is called again to obtain the second character of the escape sequence and the appropriate action is taken: If the escape sequence is '\t', '\n', or '\r', then the character is taken to be tab, newline, or carriage return, respectively. If the second character is a newline, then it is ignored, and MGET returns the result of a recursive call to itself. Otherwise, the second character is returned as the value of MGET (thus it is protected from special interpretation).

If the resulting character is not a '*' or a '%', then MGET returns that character directly. A '*' followed by a digit results in pushing a new ICB onto the stack pointing to the appropriate character string argument of the current macro call. A '*' followed by 'O', 'F', 'S', or 'R' (see Appendix I, section 3) results in a call to the C routine ANAME (which implements the NAME macro) with the appropriate arguments. When a '%' is encountered, the macro name is collected and the arguments are assembled into a 100-character buffer. The macro name and the arguments are obtained by recursive calls to MGET so that embedded macro calls are expanded; the result of expanding an embedded macro call may include commas or right parentheses without interfering with the argument structure of the macro call being processed. If the macro name is an integer, the correspondingly numbered macro definition from the machine description is used; otherwise, the macro name is looked up in a hash table containing the names of all defined macro names. If the macro is defined in the machine description, a new ICB is pushed onto the stack with LOCP pointing to the beginning of the list of pointers to character strings which represents the macro definition. Otherwise, if the macro is defined by a C routine, the C function is called and an ICB is pushed onto the stack which points to the character string returned by that function; thus references to arguments and embedded macro calls in the string returned by the C function are processed. MGET then resumes normal operation by calling GETC1. Note that the effect of a call to an undefined macro is to replace the macro call by the null string; no error messages are produced by the macro expander.

The main routine of the macro expander consists of initialization, including the setting up of the hash table, followed by a loop which calls MGET repeatedly and writes the returned character onto the output file; this loop terminates when the returned character is NULL.

5. The Error Message Editor

The error message editor is invoked as the last phase of the compiler to read from the ERROR intermediate file the error records written by the previous phases and to print error messages corresponding to those error records. The error message editor allows variable data, such as identifier

names, to be included in the printed messages. In addition, error messages of arbitrary length can be constructed from a sequence of error records; the error message editor automatically breaks long output lines so that all output lines fit within a fixed page width.

An error record is a structure containing seven integers: an error number, a line number, and five arguments. The error number selects a basic error message string which contains the fixed text of the error message and optional indicators for including variable data. An indicator is a two-character sequence beginning with a "%"; the character following the "%" defines the interpretation of the variable data which will replace the indicator when the string is printed. The variable data is specified by one or more of the arguments in the error record. The arguments are associated with the indicators from left to right; arguments are used as needed according to the interpretations specified by the indicators. The various indicators are listed below with their interpretations:

%d print the next argument as a decimal integer

%m print the string in the internal compiler table CSTORE which begins at the index specified by the next argument

%n print a string representing a node (operator) of the internal representation produced by the syntax analysis phase, as specified by the next argument

%q print a string representing the terminal or nonterminal symbol associated with the parser state specified by the next argument

%t print the source representation of the token whose TYPE and INDEX are specified by the next two arguments

%% print a "%"

Only the arguments which are referenced by the basic error message string are specified when an error record is written; the values of the remaining arguments in the record are undefined.

The line number field in the error record associates a line in the source program with the error which produced a particular error record. If a line number is given (LINENO > 0), it is printed out on a new line, followed by a colon, followed by the text specified by the error record; otherwise (LINENO <= 0), the text specified by the error record is printed on the current line. Thus an error message consists of an initial error record containing a line number followed by zero or more error records without line numbers. In this manner, an error message of arbitrary length can be constructed. For example, the message giving the current state of the parse when a syntax error has been discovered (see section 2) is constructed from the following basic error message strings:

```
"SYNTAX ERROR.  PARSE SO FAR: "
" %q"     (for each state on the parser stack)
" _"      (represents the input cursor)
" %t"     (for each of the next 5 input tokens)
```

The syntax analysis phase can produce these error messages without counting the symbols in the message or knowing their lengths because the error message editor takes care of breaking long output lines.

In addition to selecting a basic error message string, an error number represents the severity level of the corresponding error:

error number	severity
1000 - 1999	error
2000 - 3999	serious error
4000 - 5999	fatal error
6000 - 6999	compiler error

A fatal error or a compiler error will terminate the current phase, and no remaining phase (except the error message editor) will be invoked; in addition, a compiler error message is automatically preceded by the string

"COMPILER ERROR."

A serious error allows the current phase to continue execution, but all remaining phases (except the error message editor) are skipped.

The error message editor writes its output onto the standard output unit, which is normally the user's terminal in a time-sharing system or a line printer in a batch system. However, when the compiler is submitted as a batch job by a time-sharing user, this output is redirected onto an error listing file. This is accomplished by passing the argument ">>$el" to the error message editor which indicates that output to the standard output unit is to be appended onto filecode EL (the error listing file). Redirection of standard input and output is a (not necessarily portable) feature of the C run-time system, rather than of the compiler itself.

6. Invoking the Compiler Phases

The mechanisms for invoking a phase of the compiler, passing arguments to it, and returning a completion code are operating system dependent. In general, the control program will be rewritten for each system on which the compiler runs; on some systems, the control program may be replaced by a set of job control cards (see Figure 1 on page 31). The source of the compiler phases need not be changed, however; the operating system dependencies associated with the invocation of a C program are isolated in two run-time routines, the startup routine and the exit routine. The startup routine receives control from the operating system, establishes the C run-time environment, and calls the C routine named MAIN. It is the responsibility of the startup routine to take the character string arguments, which may be provided by the operating system or written on a temporary file, and arrange them as an array of character strings which is then passed as an argument to MAIN. The exit routine EXIT is called upon a return from MAIN; it may also be called directly by a C program. The exit routine closes all open files and returns control to the operating system. EXIT has one optional argument, a return code, which it communicates to the control program as a completion or abort code or by writing it onto a temporary file.

On UNIX, a phase of the compiler is invoked by calling the system routine FORK, which creates a new process, followed by a call in the new process to the system routine EXECL, which overwrites the process with the desired phase of the compiler and passes it a list of character strings as arguments. The old process waits for the execution of the compiler phase to finish by calling the system routine WAIT, which waits for the process to die and returns its completion code.

On GCOS, two methods are used to invoke a phase of the compiler from the control program, which runs in time-sharing. The first method uses a routine SYSTEM, a C-callable interface to the system call CALLSS which can invoke any time-sharing subsystem (program). The character string arguments are passed in the system teletype buffer (using the system call PSEUDO) so that to the invoked program it appears that it was invoked by a command typed at command level with those arguments. The completion code is stored (using the system call CORFIL) in the first word of the core file, a ten word buffer provided by the operating system for communication between a user's subsystems. The disadvantage of running the compiler phases in time-sharing is that the compiler phases, being large programs, can take a very large elapsed time to run. Thus this method is used only for the error message editor which prints error messages on the user's terminal.

The second method uses a routine TASK, a C-callable interface to the TASK system call, to submit a program as a special, high-priority batch activity. The elapsed time for a TASK activity is typically much lower than for the same program run in time-sharing. The character string arguments are written onto a temporary file which is read by the startup routine when in batch. The completion code is handled as follows: if there is no argument to EXIT or the argument is 0, EXIT terminates normally and TASK will return a status code of 0. Otherwise, EXIT aborts with the completion code as the abort code; the abort code is then returned in the status code by TASK.

The compiler phases can also be invoked as normal GCOS batch activities by the sequence of control cards shown in Figure 1. When these cards are submitted, IDENT and USERID cards are inserted at the beginning of the deck and the characters '=' and '%' are replaced by the user's identification and the basic component of the source file name, respectively. Thus if the user is 'B' and the source file is 'B/TEST.C', the assembly-language output will be written onto the file 'B/TEST.G' and the error messages will be written onto the file 'B/TEST.E'. The generation of the control cards and the submission of the batch job is performed by a time-sharing program (command). As the turn-around time for a normal batch job can be quite long, this version of the compiler is used only for those programs which are too large to compile using the other version of the compiler.