The C programming language and a C compiler

by R. R. Ryan
H. Spiller

In the last few years, the C programming language has become one of the most widely used languages for applications and systems software on microcomputer systems. This paper describes the C language and its history. It then presents a specific implementation of C, the Microsoft C Compiler, which runs on the IBM Personal Computer.

In 1970, Thompson, while working on the earliest version of the UNIX operating system at the Bell Telephone Laboratories, Inc., wrote an interpreter for a language patterned on BCPL, which he called B.[1] Ritchie then wrote a compiler for a derivative of B that he called C. Thompson and Ritchie wrote the next version of the UNIX operating system in the C language. Until that time, most operating systems had been written in assembly language, but Thompson and Ritchie found it so much easier working in C that they were able to add many new functions to UNIX. The first version of C was used to generate code for the then-new DEC PDP-11. The compiler was small and fast and did not attempt any ambitious optimizations. Instead, the C language provided register variables and low-level operations that allowed the programmer to guide the compiler in generating useful code.

Through the 1970s, C remained closely associated with UNIX. Most of the system is written in the C language. Compilers were written for C for use on the IBM System/370 and the Interdata 32, as well as a large number of portable programs written in the C language.[2] AT&T began to use C to develop their switching systems, and eventually it became the main development language at Bell Laboratories. In the mid-1970s, UNIX was introduced into the academic environment at a number of universities, and the popularity of C started to spread. By the late 1970s, C compilers were becoming available for a large number of processors and operating systems, and papers began to appear comparing C with other languages for various applications.[3-5]

In the original language, there was little in the way of type checking and few restrictions as to type. The C language was invented by experienced programmers for their own use, and it provided them with a great deal of power but very little protection from data-type errors. A program called "lint"[2] for checking types in C programs was the only way to provide such protection. As the language has evolved, more attention has been paid to types and to type checking within both the language and the compilers, but the philosophy of C remains unchanged. Even as the language has incorporated stronger type checking, there has continued to be a mechanism to override its restrictions.

At first, the typical C user was a systems programmer who needed to be close to the machine and who

required the efficient code generation of assembly language. At the same time, such programmers wanted the fluency of expression inherent in a high-level language. However, they were doing their development on the target machine, which was a min-

C is a popular choice of applications programmers, although it is rarely one's first programming language.

icomputer and thus not large enough to support large optimizing compilers. Therefore, the early C compilers saved space by allowing the programmer to do some of the optimization.

The 16-bit personal computers of today are about the same size as the minicomputers of the early days of C. Application writers working on personal computers are under very much the same constraints as systems programmers using C in the early 1970s. Because of this, C has proved to be a popular choice among applications programmers, although it is rarely one's first programming language. Experienced programmers appreciate the ability to write powerful programs economically in C.

In 1983, an ANSI standards committee (X3J11) for C was formed. Kernighan and Ritchie[6] had earlier written a reference manual, which was the closest thing to a standard. However, the manual was not precise and did not contain later features in the language, nor did it document the standard library. The major job of the committee has been to codify common understanding, because the C language has had remarkably few local extensions added over the years. Among those few extensions are argument checking, structure copying, and an enumerated type.[7] There has also been an effort by UNIFORUM, a UNIX users' organization, to create a standard for the C library.[8] Although this standard is oriented toward UNIX, most of it is sufficiently general that the ANSI committee is using it as a basis for their library standard.

Concepts of the C language

The C programming language[6] is expression-oriented, with a rich set of operators and a block structure, encouraging well-structured programs. It includes constructs for decision making (**if** statements), several types of looping (**for**, **while**, and **do-while**), and selection of cases (**switch**). In keeping with the spirit of low-level languages, there is also support for **goto**.

The C language is relatively low-level, providing a set of operators that work on the same kinds of objects that most computers do. This allows the programmer to work close to the underlying machine. Yet C is of a sufficiently high level to make programs portable across a wide variety of machines. The language views the underlying machine as a single monolithic array, allowing objects to be referenced anywhere in that array. Pointer arithmetic and array subscripting both manipulate the same kind of object—a memory address.

The fundamental data types that C supports are characters, integers of several sizes, and floating point numbers of both single and double precision. Pointers can be declared to all data types, including derived data types and functions. New data types can be derived through the use of arrays, pointers, structures, unions, and functions.

C is a relatively simple language that provides few mechanisms for dealing directly with complex data structures such as strings, lists, and arrays. Similarly, there is no direct language support for dealing with the environment (no input-output, no memory allocation, etc.). Instead, these capabilities are provided through function calls. In the evolution of the C language, a particular set of function calls (the so-called *standard library*) has come to be considered part of the language. But because the language compiler does not recognize anything special about these function calls, a programmer can replace standard functions with special implementations if necessary.

Another powerful feature of the language is a text processor, called the *C preprocessor*, which allows the use of macros, conditional compilation, and the inclusion of other text files. These features facilitate the writing of C programs that are portable to a variety of machines and operating systems.

The simplicity of the language has allowed compilers to be written easily and quickly. The expression

orientation of the language and the rich set of operators enable the programmer to create efficient programs using simple compilers.

Two examples

C is attractive to systems programmers because of the number of low-level operators and data types provided. One task these low-level operators are especially suited to is manipulating linked lists. The following are two somewhat dense examples illustrating a number of the operators used in initializing and scanning a linked list. The function is presented in C, followed by a discussion of the operators involved. Note that anything occurring between /* . . . and . . . */ is a comment.

Example 1. The following are some basic functions of the C language.

```
/* Preprocessor directives—replace all occurrences
   of */

#define SIZE 100            /* 'SIZE' with 100 */
#define NULL 0              /* 'NULL' with 0 */

struct thing {              /* Declare a structure
                                 type, 'thing', */
   struct thing *link;      /* consisting of a pointer
                                 to a 'thing' */
   char val;                /* and an 8-bit signed
                                 integer */
};

struct thing Table[SIZE];   /* declare an array of
                                 'things' */
struct thing *Freething;    /* a pointer to a thing */

/* initialize a freelist of
   things */
init()
{
    register struct thing   /* declare a local register
      *p;                       pointer variable */

    for(p = Table; p < &Table[SIZE]; p+=1)
                            { /* make each element */
        p - >link = p-1;    /* point to the previous
                                 element */
        p - >val = 0;
    }
    Freething = p-1;        /* initialize freelist head */
    Table[0].link = NULL;               /* and tail */
}
```

A register declaration instructs the compiler that the variable should be placed in a register if possible. Registers are allocated in the order declared as long as they are available; otherwise, they are treated as local variables. Our implementation for the Intel 8086/80286 provides two register variables, SI and DI; other processors may have more or fewer. The code produced using register variables is almost always smaller and faster than that using ordinary local variables.

Code produced using register variables is almost always smaller and faster than that using local variables.

A for-loop is of the form

```
for (expression1; expression2; expression3)
    BODY OF LOOP
```

which can be expressed in a pseudocode as follows:

```
evaluate expression1
while (expression2 is TRUE)
    BODY OF LOOP
evaluate expression3
end-of-loop
```

The for-loop in the initialization example just given illustrates several features of C. The first clause sets a pointer p to point to the base of the array Table. In C, the name of an array is synonymous with the address of the first element. A pointer is a variable containing an address.

Following the first semicolon is the for-loop's end condition. In this case, the value of the pointer is being tested against the address of the end of the array. Following the second semicolon there is an expression to be evaluated as the last action each time through the loop. Any expression can be put there.

In Example 1, expression3 of the for-loop uses the += operator, which has the following form:

```
expression1 += expression2
```

is equivalent to

```
expression1 = expression1 + expression2
```

with the exception that expression1 is evaluated only once; thus any side effects occur only once. In this case, the += operator is adding a constant value to the pointer. In C, pointer arithmetic is scaled to the size of the object pointed to. Because we are adding to a pointer to a "thing" in the example, the constant 1 is scaled to the size of the "thing."

Pointer arithmetic is machine independent.

Doing this scaling based on pointer type has an interesting and useful effect. Because an array name is equivalent to a pointer to the base of the array, and adding an index to a pointer causes the index to be scaled by the element size, the resulting operation is the same arithmetic used for array subscripting. Incidentally, this means that array subscripting is commutative (i.e., array [i] == i[array]) because addition is commutative. This makes pointer arithmetic machine independent.

The braces {...} delimit a compound statement that is to be executed each pass through the loop. The first statement assigns the address of the previous element to its own link field, thereby taking advantage of the pointer scaling operation. The -> operator references a field of the structure pointed to.

The second statement inside the braces initializes the integer field. The two statements after the for-loop clean up the boundary conditions. This is not intended to be a stylistically "nice" routine but rather a dense illustration.

Example 2. The following example illustrates further expressions, using the same data structure as that in Example 1:

```
/* Return the nth element of the list, or NULL if
 * there aren't that many. The function is de-
 * clared to return a pointer to a 'thing'
 */
struct thing *nth(list, n)
    register struct *list;
    register int n;
{
    while(list && n--) /* while there is list left,
                          count out elements */
      list = list->next;
    return(n>=0 ? NULL : list); /* use count to de-
                                   cide if too many
                                   */
}
```

In C, the value 0 is treated as a boolean false; anything else is true. Thus, the statement while(expr) is the same as the statement while(expr != 0). (The symbol != means "not equals.")

The && operator is a short circuit logical AND. If the condition before the && is false, the truth value is known and evaluation is stopped. This has a benefit other than performance in that it can be used to protect illegal evaluations from being made. Thus, for example, if(list && list->val... does not try to dereference a null pointer.

The expression n-- is a use of the post-decrement operator that has the same effect as a function containing

```
temp = n

n = n - 1

return(temp)
```

Thus, in Example 2, the value tested is the value before the variable is decremented. Like many of the special low-level operators in C, this gives the compiler a very good hint about generating good code, especially because many machines have an addressing mode that does exactly this. An idiom which is often seen is the following:

```
while(*p++ = *q++)
        ;
```

This copies a null-terminated string. On some machines, it is easy for the compiler to recognize this idiom and produce an optimal machine instruction sequence.

The return statement sets the return value of the function. In this case, the return statement is using

Sharing compiler technology across languages and processors both lowers the cost and increases the overall quality.

a ternary conditional operator, a C language construct of the following form:

```
expression1 ? expression2 : expression3
```

which is equivalent to

```
if (expression1 is TRUE)
    evaluate expression2

else
    evaluate expression3
```

The result of the whole expression is the value of whichever expression has been evaluated.

Microsoft C compiler

The Microsoft C compiler is part of a project of the Microsoft Corporation to create a set of common compiler phases for several languages and machines. Like many language producers, we realized that writing good compilers is expensive and that sharing compiler technology across languages and processors both lowers the cost and increases the overall quality. Thus, we designed an intermediate language (IL) that allows language processors to communicate with a machine-independent optimizer, which in turn communicates through another IL to the machine-dependent portions of the compiler. At present we have language front ends for C, Pascal, FORTRAN, and BASIC. Code generators have been written for the Intel 8088/8086/286,[9] as well as for other machines. Written in C, the compilers are very portable and run on IBM PCs and other systems.

Because C continues to be an evolving language, the first question we asked was which C we should implement. We started with the UNIX System-V C compilers, which we used as a base language definition. When the ANSI standards effort began, we reviewed the extensions proposed by the committee and incorporated the ones that seemed likely to become part of the eventual standard.

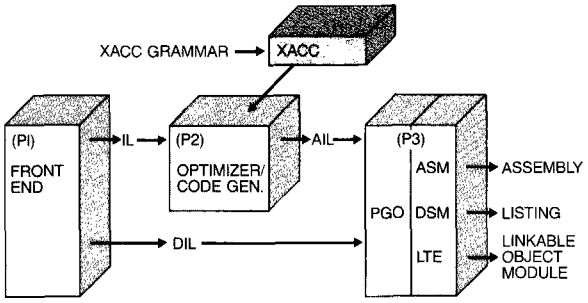
It has been important to keep the compilers consistent across machines and operating systems and to provide strong cross-development tools. For example, the relocatable object format is identical for PC DOS and for XENIX on the 8086 and 286.

Memory models. One of the difficulties of creating a C compiler for the 8088/8086/286 is the segmented architecture of the processor. The fact that the C language views the underlying machine as a flat address space creates a conflict. The usual approach to this problem is to construct the following memory models to approximate the C assumptions:

- *Small.* All code and data addresses are 16 bits. This limits the program to two 64K-byte segments, one for code and one for data.
- *Medium.* Code addresses are 32 bits, but data addresses are 16 bits. This model allows an arbitrary number of code segments, but it is limited to a single data segment.
- *Compact.* The inverse of the medium model, the code addresses of the compact model are 16 bits and data addresses are 32 bits. A single code segment can access an arbitrary number of data segments.
- *Large.* Both code and data addresses are 32 bits, and both can have an arbitrary number of segments.

The major problem with pure memory models is that a program must pay a severe performance penalty to access large quantities of code or data. It is more expensive to manipulate 32-bit pointers than 16-bit pointers. The problem is inherent in the architecture, which is really that of a 16-bit machine. For many applications, the ability to address large amounts of memory is not needed for the whole program but only for selected portions. To address this need, we have added the following two key words to the language:

Figure 1 General architecture of the C compiler



- *near* denotes an item with a 16-bit address.
- *far* denotes an item with a 32-bit address.

Regardless of the memory model, an object declared with a near or far attribute is given the appropriate allocation. The programmer now has a way to escape from the memory model. Realistically, programmers tend to compile programs using the small or medium model and then use far data pointers to access the few data structures that need to extend beyond the size of a segment. The converse is less useful, but it is nonetheless available to provide near calls to local procedures in middle-model programs or access to near data in large-model programs.

General architecture. The Microsoft C compiler consists of three phases—P1, P2, and P3—as shown in Figure 1. The P1 phase or front end reads the source program and does the C language preprocessing. This includes macro substitution, conditional compilation, and inclusion of named files. P1 then translates the program into expression trees, doing semantic analysis, checking for syntax and type correctness, and writes an intermediate language (IL) temporary file. This intermediate language is both machine- and language-independent and contains information about such things as control structures, expressions, symbols, and data initialization. P1 also emits DIL, which contains data-type information to be used by a symbolic debugger.

P2 is the first pass of the back end. It translates the intermediate language (IL) to a form of machine instruction. It first reads the IL and builds expression trees. Thus P2 assigns specific storage to data and does various optimizations: constant folding, strength reduction (e.g., converting a multiplication by a power of two into a shift), type optimizations (e.g., changing a 32-bit addition to a 16-bit addition, if that is the precision required in the result), common subexpression elimination, and value propagation. Aho and Ullman[10] describe these optimizations in more detail. After P2 has done these optimizations, the expression trees are processed by the code generator. This part of P2 is generated automatically from a pattern language processor called XACC and is described further in the next section. P2 translates the optimized expression trees into Assembly Intermediate Language (AIL), also described further in the next section. AIL is very similar to assembly language but in a form that is more useful to the rest of the compiler.

The AIL is read in by P3, the last part of the back end. Based on command line switches, P3 optionally passes the code through the Post-Generation Optimizer (PGO), which does further code improvement. The PGO is described in a later section of this paper. After it has completed its optimizations, P3 builds instructions and emits one or more of the following:

- *Link text* for the 8086/80286. This relocatable format is based on the Intel Object Module Format (OMF).
- *Assembly language source* for the code being generated.
- *A listing* that looks like the listing file of the typical assembler. The listing contains the offsets and binary values of the emitted instructions, as well as the assembly language.

Phase P1, the early phases of P2, and much of P3 are fairly conventional. Aho and Ullman[10] and many of the other compiler construction textbooks are good references for most of these processes. The Code Generator and the PGO, which are more interesting, are now presented in greater detail.

Code generation. The code generator translates intermediate language (IL) into an assembly language (AIL). The IL consists of trees representing the various expressions from the source program, and the AIL is a set of instructions and addressing modes for the target machine. Because the translation is usually complex, writing high-quality code generators is a difficult task. The problem is largely one of recognizing patterns in the input representation and translating them to the output representation in an optimal way. Glanville[11] observed that since the code generator, like the front end, is a translator, much of the theoretical work on pattern matching for front ends can be applied to the code generator.

The Glanville approach uses a tool called a LALR parser generator, which takes a formal description of the input language and produces a parsing program

that recognizes patterns in the input language and makes them available for translation.[10] When a pattern is recognized, it is a simple matter to write out the appropriate output language. Much of the power of the Glanville technique comes from the ability to have the parser recognize arbitrary patterns that correspond to machine instructions.

However, a number of workers have discovered that register targeting is a problem with this approach and have used various techniques to deal with it.[12-15] That is, if an intermediate calculation is needed for some purpose—say as a pointer—it may be necessary to perform the calculation in a special-purpose register. Glanville's approach recognizes patterns by collecting smaller patterns into larger ones (i.e., in a bottom-up manner). Thus, it does not recognize the context in which a pattern is used.

A pattern can consist of terminal symbols, other productions, conditions to be met, or actions to be taken.

Our approach also uses a parser built by a parser generator, but the matching process is steered by the eventual use of the pattern, in addition to the component patterns (i.e., a top-down approach). We now illustrate how our code generator works by stepping through the translation of a C program fragment.

The pattern description language is of the following form:

```
production : pattern
           | optional alternate pattern
                  :
                  :
           ;
```

A pattern can consist of terminal symbols, other productions, conditions to be met, or actions to be taken. If the first pattern fails to match, the second is tried, and so on. A pattern description is processed by the XACC pattern language processor, which produces a parser that is part of P2. In the following example, each time a pattern is recognized a code template is associated with the pattern. This template has the following three parts:

- CODE represents some number of machine instructions (possibly zero). Code templates look much like assembly language, but have substitution macros (indicated in the example by a leading % symbol) and other operators imbedded in them.

- DEFER is analogous to the return value of a function. The deferred strings allow us to build up a single machine instruction out of component patterns. As an example, an instruction typically uses the deferred strings of its subpatterns for operand addressing modes. The addressing modes have, in turn, been built up from simpler components such as register names and constants.

- ATTRIB contains miscellaneous attributes of the template.

  1. The first field contains the number of child templates or subpatterns.

  2. The second field consists of the following two parts:

     Rewrite rule. Whenever the actions of a template are complete and AIL has been generated, the original expression tree is rewritten to reflect the current state, most commonly as REREG (in a register) or REFAD (an effective address).

     Path. When machine code is eventually emitted, the lower regions of the tree come out first. Each template in the code sequence may have some restrictions on allowed registers. This attribute, termed the *path*, is used to pass register constraint information downward. These constraints are accumulated so that the eventual selection of a register satisfies the tightest constraint on its use along a path.

  3. The third field contains operators that determine the order in which code is emitted. This is described in more detail later in this paper.

  4. The remaining fields contain register allocation information. Each of these fields has three subfields:

Link attaches the other two subfields to a particular child template.

Constraint places restrictions on which registers may be selected. CAREG, for example, states that there must be an address register. Note that the child template can further propagate this constraint using the path parameter previously described.

Allocation attributes declare what to do with a register: NEW to allocate a new register; FREE to free all registers used by the subpattern or others.

Usually there are only one or two of these subfields that are nonzero. The following is the example for which we have been setting the stage. We first give an example of an XACC input grammar. Although it is not complete, it is sufficient to compile the C expression that follows it.

```
start    : ASSIGN address reg
             {
             CODE("mov %bw %1, %2");
                                    /* template 1 */
             DEFER ("");
             ATTRIB("2, 0, 0, 1|0|FREE, 2|0|FREE,
               0,0");
             }
           ;
reg      : EXTRACT address
             {
             CODE("mov %bw %1, %2");
                                    /* template 2 */
             DEFER("%1");
             ATTRIB("1, REREG|1, SU_SCR, 0|0|NEW,
               1|0|FREE, 0,0");
             }
           | PLUS reg constant
             {
             CODE("add %bw %1, %2");
                                    /* template 3 */
             DEFER("%1");
             ATTRIB("1, REREG|1, SU_SCR, 0|0|0,
               1|0|FREE, 0,0");
             }
           ;
address  : NAME
             {
             CODE("");              /* template 4 */
             DEFER("[%S]");
             ATTRIB("0,0,0, 0,0,0,0");
             }
           | PLUS addr_reg constant
             {
             CODE("");              /* template 5 */
             DEFER("[%reg1 + %2]");
             ATTRIB("2, REFAD|0, SU_ADR,
               1|CAREG|0, 2|0|0, 0,0");
             }
           ;

constant: CONSTANT
             {
             CODE("");              /* template 6 */
             DEFER("%C");
             ATTRIB("0,0,0, 0,0,0,0");
             }
           ;
addr_    : EXTRACT address
  reg
             {
             CODE("mov %bw %1, %2");
                                    /* template 7 */
             DEFER("%1");
             ATTRIB("1, REREG|1, SU_ADR,
               0|NEW|CAREG, 1|FREE, 0,0");
             }
           | reg
           ;
```

The macros used in the example are defined as follows:

%1	Replace with the results of the first link field in the ATTRIB. This could be either the DEFER string of a subpattern or the result of allocating a new register.
%2	Similarly, replace with the results of the second link field.
%C	Substitute the value of the constant in the associated expression tree.
%S	Substitute the symbol name from the associated expression tree.
%reg1	Replace with the register name returned in the first link field.

Here is an example C expression to be compiled:

```
i = p->x + 3;
```

This expression means to get the value of the x field of the structure pointed to by p, add 3, and assign to the variable i.

After P1 and the optimizer stage of P2 are done, the expression tree looks as shown in Figure 2.

Note that the NAME node represents a memory address. The EXTRACT node is used to get the contents of that address. The parser generated from the pattern grammar given previously tries to match this tree by doing a preorder or forward Polish walk of this tree. This is written as follows:

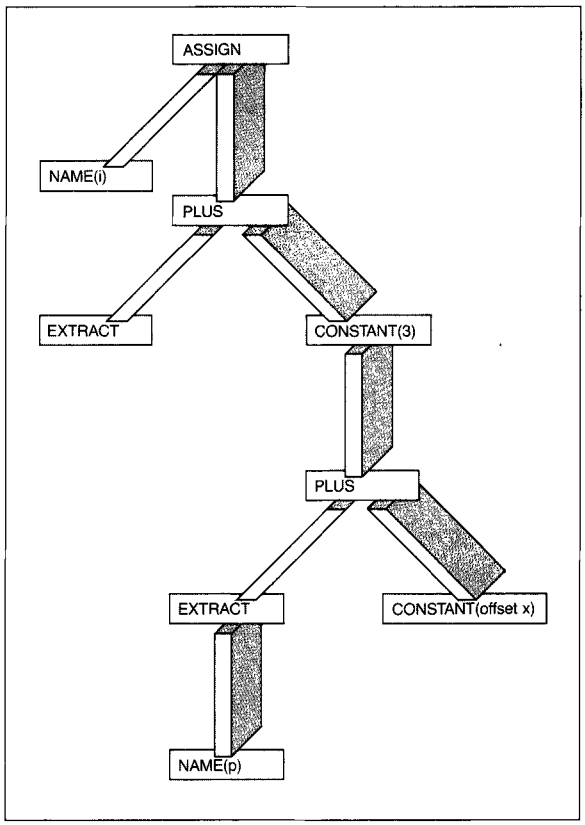
ASSIGN NAME(i) PLUS EXTRACT PLUS EXTRACT
NAME(p) CONSTANT(offset x) CONSTANT(3)

In the pattern to be matched, *start* consists of ASSIGN (which matches), and two subpatterns: address and reg. The *address* pattern matches the NAME node. Similarly, the rest of the input matches the *reg* pattern and its associated subpatterns. Each time a pattern is recognized, the associated template (if it exists) is pushed onto a stack. When the entire matching process is complete, a new tree is created parallel to the original expression tree. Figure 3 shows the parallel tree for this example. The instructions from the CODE fields are used to identify all the operands. The number in parentheses is the number of the template producing the node.

To generate AIL from this parallel tree, we must determine the optimal order in which to walk the tree, and we must assign registers where required. The first walk of the parallel tree analyzes the order operators (the third field in each ATTRIB record). These operators have the prefix SU_, after Sethi and Ullman, who created an ancestor of the technique;[16] such operators are used to decide the order in which to walk the tree. The ordering heuristic deals with overlapping and conflicting register classes. For each node in the parallel tree, a *held vector* (the registers left busy from this calculation) and a *used vector* (the total set of registers used by this entire subtree) are computed on the basis of the SU_ operator. Each component of the vector contains the number of registers held or used in that register class by the expression. The sum of held and used registers in each class predicts where the register allocator will run out of registers and *spill code* will have to be generated. The ordering phase attempts to minimize the number of spills by considering whether there are fewer spills in first walking the left side of the tree in Figure 3, then the right side, or vice versa. In the example, there are no conflicts, so that any order is as good as any other order.

After deciding the order in which to generate code for the parallel tree nodes, we do another walk in the established order, passing register constraints from the ATTRIB fields down the tree. These constraints

Figure 2 Example expression tree after P1 and P2 processing

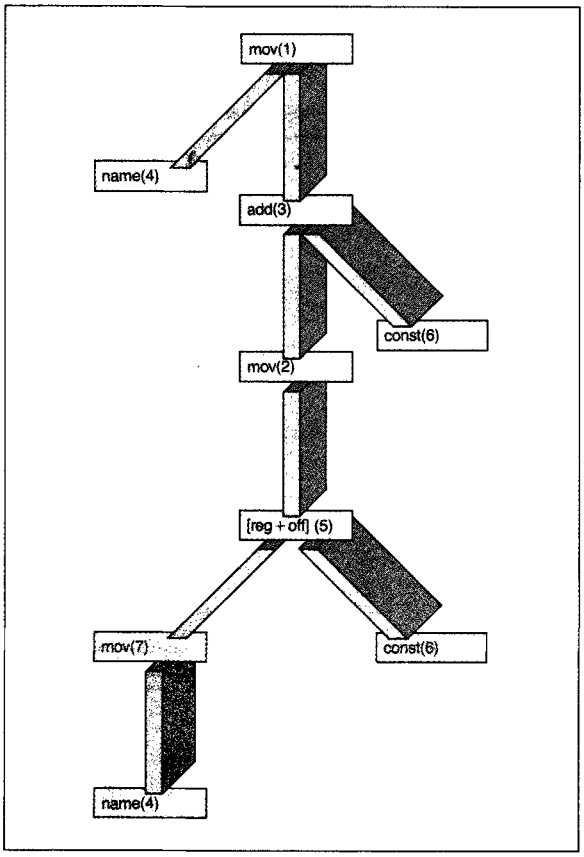


are accumulated so that the ultimate selection of a register satisfies the strictest constraint used in the expression. In the example, the addressing mode template (5) constrains its first child to CAREG (Constrain to Address REGister), which, on the 8086, forces one of BX, SI, or DI to be chosen. If there had been other constraints, say, that the register must be convertible to a byte register, we would have restricted it further to the intersection of the two sets. That is, the intersection of {AX,CX,DX,BX} and {BX,SI,DI} is {BX}.

As the leaf nodes are reached, we back out of the tree, allocating registers on the fly according to the accumulated constraints and emitting code to the AIL stream. The original expression trees are rewritten according to the rewrite field of the ATTRIB in the template in preparation for potential spills (discussed below).

Here is the example in C again:

Figure 3 New expression tree after completion of the matching process




```
i = p ->x + 3;
```

and the code produced is the following:

```
mov   bx,p        move the contents of p into
                    BX
                  template 2 with operand from
                    template 4

mov   ax,[bx+x]   move the contents of address
                    BX+X into AX
                  template 2 with operand from
                    template 5 (which was built
                    from 2 and 6)

add   ax,3        add 3 to AX
                  template 3 with operands
                    from 2 and 6

mov   i,ax        move the contents of AX into i
                  template 1, with operands
                    from 4 and 3
```

When there are unresolvable register conflicts, it is necessary to spill an intermediate value from the demanded register to another location in order to proceed with the calculation. On machines like the 8086, it is often more effective to spill to a different register than to spill to memory. Regardless of the choice, the spill sufficiently changes the pattern of code to be generated that a reanalysis is in order. Thus, we emit code to make the required register available, rewrite the original tree to represent the current state of the evaluation, and run the whole grammar/order/register allocation algorithm again.

The Post-Generation Optimizer. The Post-Generation Optimizer (PGO) is based largely on the FINAL phase of the Bliss compiler.[17] The PGO performs a variety of optimizations that the code generator cannot do or does not do because the PGO can do a better job. These include peephole optimizations, which involve replacing sequences of code with more efficient sequences that produce the same result. The PGO also performs a simple flow analysis to eliminate redundant register loads. Branches are shortened where possible, and several simple code movement optimizations are applied. One of the more interesting of these code movement optimizations, which is called *cross-jumping*, involves sharing the instructions of a common sequence before a common label.

Consider the following C statements: if (x) then x = y+z; else x = w+z. P2 generates the following 8086 instructions. The comments after each line describe each instruction rather than explaining the general flow.

```
        cmp   x,0  ; compare x and 0
        je    L1   ; branch if x == 0
        mov   ax,y; load y into register ax
        add   ax,z  ; add z to register ax
        mov   x,ax; store register ax into x
        j     L2   ; unconditional branch
L1:   mov   ax,w   ; load w into register ax
        add   ax,z  ; add z to register ax
        mov   x,ax; store register ax into x
L2:
```

Observe that the last two instructions of each path

```
add   ax,z    ; add z to register ax
mov   x,ax    ; store register ax into x
```

are the same. Cross-jumping takes advantage of this and rewrites the whole sequence as

```
        cmp   x,0 ; compare x and 0
        je  L1    ; branch if x == 0
        mov   ax,y; load y into register ax
        j   L2    ; branch
L1:  mov   ax,w   ; load w into register ax
L2:  add   ax,z   ; add z to register ax
        mov   x,ax; store register ax into x
```

Because our target machines are small, our objective has been to concentrate on generating small code. Small code is usually fast code. Cross-jumping does not change execution speed, but it reduces code size. Such sequences are surprisingly common in generated code.

P1 passes line numbers to P2, which passes them on to P3. This allows the creation of code offset-line-number tables in the object module. Data allocation information is passed from P2 to P3 in the AIL, and P3 merges it with the type information from the DIL to create a symbol table in the object module. A symbolic debugger capable of using this information is currently being studied.

Concluding remarks

In the past two years, there have been over ten new C compilers offered for the PC. The availability of so many compilers has made more people aware of the language. The hardware requirements are basic. A programmer can adequately use microsoft C on the PC with two floppy disks and as little as 192K bytes of memory. For a realistic development environment, a minimum of 256K bytes of memory and a hard disk are preferable. A growing number of development tools are becoming available for use with C—special-purpose libraries, symbolic debuggers, and program-oriented editors. C has proved to be a fine systems and applications language for small computers. The greatest advantage of C stems from its original concept, which is to give the programmer both low-level control of the machine and high-level expression of programming concepts.

Cited references

1. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "The C programming language," *The Bell System Technical Journal* **57**, No. 6, Part 2, 1991–2020 (July–August 1978).
2. S. C. Johnson and D. M. Ritchie, "Portability of C programs and the UNIX System," *The Bell System Technical Journal* **57**, No. 6, Part 2, 2021–2048 (July–August 1978).
3. A. Feuer and H. Narain, "A comparison of the programming languages C and Pascal," *ACM Computing Surveys* **14**, No. 1, 73–92 (March 1982).
4. A. Feuer and N. Gehani, *Comparing and Assessing Programming Languages: Ada, C, and Pascal*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1984).
5. J. Gilbreath, "A high-level language benchmark," *Byte Magazine* **6**, No. 9, 180–198 (September 1981).
6. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1978).
7. *Draft Proposed Standard—Programming Language C*, ANSI X3J11 Language Subcommittee, L. Rosler, Chairman, Computer and Business Equipment Manufacturers Association, Washington, DC (1984).
8. *Proposed Standard, UNIFORUM*, /usr/group Standards Committee (August 31, 1983).
9. *Intel MCS-86 User's Manual*, Intel Corporation, Santa Clara, CA (February 1979).
10. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley Publishing Co., Reading, MA (1977).
11. R. S. Glanville and S. L. Graham, "A new method for compiler code generation," *Proceedings of the ACM Principles of Programming Languages Conference* (January 1978), pp. 509–514.
12. P. Aigrain, S. L. Graham, R. R. Henry, M. K. McKusick, and E. Pelegri-Llopari, "Experience with a Graham-Glanville style code generator," *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction* (June 1984), pp. 13–24.
13. T. W. Christopher, P. J. Hatcher, and R. C. Kukuk, "Using dynamic programming to generate optimized code in a Graham-Glanville style code generator," *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction* (June 1984), pp. 25–36.
14. J. Crawford, "Engineering a production code generator," *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction* (June 1982), pp. 205–225.
15. M. Ganapathi and C. N. Fischer, "Description driven code generation using attribute grammars," *Proceedings of the ACM Principles of Programming Languages Conference* (January 1982), pp. 108–119.
16. R. Sethi and J. D. Ullman, "The generation of optimal code for arithmetic expressions," *Journal of the ACM* **17**, No. 4, 715–728 (October 1970).
17. W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler*, Elsevier North-Holland Publishing Co., New York (1975).

General references

Byte Magazine **8**, No. 8 (August 1983), whole issue.

P. M. Chirlian, *Introduction to C*, Matrix Publishers, Beaverton, OR (1984).

A. R. Feuer, *The C Puzzle Book*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1982).

M. Ganapathi, C. N. Fischer, and J. L. Hennessy, "Retargetable compiler construction," *ACM Computing Surveys* **14**, No. 4, 573–592 (December 1982).

L. Hancock and M. Krieger, *The C Primer*, McGraw-Hill Book Co., Inc., New York (1982).

S. C. Johnson, *YACC—Yet Another Compiler Compiler*, Report No. 32, Bell Telephone Laboratories, Inc., Murray Hill, NJ.

T. Plum, *Learning to Program in C*, Plum Hall, Inc., Cardiff, NJ (1983).

J. J. Purdum, *C Programming Guide*, Que Corporation, Indianapolis, IN (1983).

T. O. Sidebottom and L. A. Wortman, *The C Programming Tutor*, R. J. Brady Co., Bowie, MD (1984).

R. J. Traister, *Programming in C for the Microcomputer User*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1984).

Z. T. Zahn, *C Notes, A Guide to the C Programming Language*, Yourdon Press, New York (1979).

Reprint Order No. G321-5236.

Ralph R. Ryan *Microsoft Corporation, 10700 Northrup Way, Box 97200, Bellevue, Washington 98009.* From 1978 to 1981, Mr. Ryan worked as a contract software engineer in the area of distributed databases. In 1981, he worked for the Boeing Aerospace Corporation on a portable compiler project. Mr. Ryan joined the Microsoft Corporation in 1982 and is currently manager of the Compiler Technology (CMERGE) group. Mr. Ryan received a B.S.E.E. from Rutgers University in 1977 and an M.S.E. in electrical engineering from Princeton University in 1978.

Hans Spiller *Microsoft Corporation, 10700 Northrup Way, Box 97200, Bellevue, Washington 98009.* From 1978 to 1981, Mr. Spiller supported and developed compilers at the Data General Corporation, including compilers for Pascal and DG/L compilers for the Nova, Eclipse, and Eagle (MV/#000 series). He joined the Microsoft Corporation in 1981. There he has developed the XENIX language products, including C compilers, assemblers, and debuggers for the 68000, the Z8000, and the 8086. He is now responsible for the CMERGE 8086/286 code generator. Mr. Spiller received an A.B. in computer science from the University of California at Berkeley in 1978.