A Code Generation Interface for ANSI C

Christopher W. Fraser

AT&T Bell Laboratories, 600 Mountain Avenue,
Murray Hill, NJ 07974

and

David R. Hanson

Department of Computer Science, Princeton University,
Princeton, NJ 08544

Research Report CS-TR-270-90
July 1990
Last Revised September 1991

Abstract

1cc is a retargetable, production compiler for ANSI C; it has been ported to the VAX, Motorola 68020, SPARC, and MIPS R3000, and some versions have been in use for over two years. It is smaller and faster than generally available alternatives, and its local code is comparable. This report describes the interface between the target-independent front end and the target-dependent back ends. The interface consists of shared data structures, a few functions, and a dag language. While this approach couples the front and back ends tightly, it results in efficient, compact compilers. The interface is illustrated by detailing a complete code generator that emits naive VAX code.
1 Introduction

1cc is a retargetable compiler for ANSI C [1]. It has been ported to the VAX, Motorola 68020, SPARC, and MIPS R3000. It emits code that is comparable with that from other generally available C compilers, but it runs up to twice as fast and is about half the size [5]. 1cc is in production use at Princeton University and AT&T Bell Laboratories.

This paper describes the interface between the target-independent front end and the target-dependent back ends.\(^1\) Good code-generation interfaces are hard to design. An inadequate interface may force each back end to do work that could otherwise be done once in the front end. Annotating frequently referenced variables for assignment to registers is an example. If the interface is too small, it may not encode enough information to exploit new machines thoroughly. If the interface is too large, the back ends may be needlessly complicated. These competing demands require careful engineering, and re-engineering as new targets expose flaws. This paper reports the results of such experience.

The interface is illustrated with a sample code generator for the VAX. This code generator emits naive code; i.e., it uses only the ‘RISC subset’ of the VAX instruction set. It is nevertheless complete: when used with a conforming preprocessor and library, the compiler with this code generator passes the conformance section of Version 2.00 of the Plum-Ihall Validation Suite for ANSI C, except that it does not detect overflow in floating constants. The production versions of 1cc use the interface described here, but use back ends in which instruction selection and optimization are generated automatically from a compact specification [9].\(^2\)

Unlike other interfaces, 1cc’s interface is not a monolithic intermediate language [9]. Such interfaces promote decoupling between the front and back ends, but sacrifice compiler performance [8]. With 1cc’s interface, the front and back ends are tightly coupled; this approach yields efficient, compact compilers, but can complicate maintenance because changes to the front end may affect the back ends. This complication is less important for standardized languages like ANSI C because there will be few changes to the language.

The interface consists of a few shared data structures, 19 functions, most of which are very simple, and a 36-operator dag language, which encodes the executable code from a source program. The dag language corresponds to the ‘intermediate language’ used in other compilers, but it is smaller than typical intermediate languages. The functions, which can be implemented as true functions or as macros, are listed in Appendix A and are described in the sections below.

The front and back ends are clients of each other. The front end calls

\(^1\)References [4] and [6] were drawn from earlier versions of this report. This report is slightly more detailed and corresponds to the latest release of 1cc.

\(^2\)The 1cc front end and a sample code generator are available for anonymous ftp from princeton.edu. The file README in the directory pub/1cc gives details.
on the back end to generate and emit code. The back end calls on the front end to perform output, allocate storage, interrogate types, and manage nodes, symbols, and strings. These front-end services are performed by functions that are summarized in Appendix B.

2 Configuration

Target-specific configuration parameters specify the widths and alignments of the basic datatypes and optionally define conditional compilation flags. They are defined in a 'header' file, config.h, which is included when a target-specific ICC is compiled.

The sample type metrics and conditional compilation flags are defined as follows. Target-specific components of the shared data structures are defined in Sections 3 and 5.

#define VAX

/* type metrics: size, alignment, constants */
#define CHAR_METRICS 1,1,0
#define SHORT_METRICS 2,2,0
#define INT_METRICS 4,4,0
#define FLOAT_METRICS 4,4,1
#define DOUBLE_METRICS 8,4,1
#define POINTER_METRICS 4,4,0
#define STRUCT_ALIGN 1

#define LEFT_TO_RIGHT /* evaluate args left-to-right */
#define LITTLE_ENDIAN /* right-to-left bit fields */
#define JUMP_ON_RETURN 0

Type metrics are triples that give the size and alignment of the type in bytes, along with a flag indicating whether or not constants of that type can appear in instructions. A 1 indicates that the constants cannot appear in instructions; the front end generates variables to hold them. The size of a type must be a multiple of its alignment.

Both the size and alignment for characters must be 1. Unsigned and long integers are assumed to have metrics identical to integer, and long doubles are assumed to have metrics identical to double. The front end correctly treats all of these types as separate, however. POINTER_METRICS apply to all pointer types, and pointers must fit in unsigned integers. The alignment of a structure is the maximum of the alignments of its fields and STRUCT_ALIGN.

If JUMP_ON_RETURN is non-zero, the front end generates a jump to a generated label for each return statement and defines this label at the end of each function. Similar action is taken if the compiler is asked to generate data for a debugger,
because some debuggers assume a common exit point. Since the VAX does
returns with one instruction, \texttt{JUMP\_ON\_RETURN} is defined as 0.

By default, the front end generates code that evaluates function arguments
from right to left; defining \texttt{LEFT\_TO\_RIGHT} yields the opposite order. A se-
quence of bit fields is laid out from left to right in one or more unsigned values;
defining \texttt{LITTLE\_ENDIAN} yields a right-to-left layout. Defining \texttt{LITTLE\_ENDIAN}
for the sample makes the code compatible with existing VAX C compilers. The
standard permits either argument evaluation or bit-field layout order, however.

The front end contains a few target-specific operations. These are protected
by conditional compilation on \texttt{VAX}, \texttt{MIPS}, \texttt{MC}, \texttt{SPARC}, etc. Thus, \texttt{VAX} is defined
above.

\section{2.1 progbeg and progend}

During initialization, the front end calls

\begin{verbatim}
void progbeg(int argc, char *argv[])

argv[0..argc-1] point to those program arguments that are not recognized by
the front end, e.g., target-specific options. Typical implementations of progbeg
process such options and initialize themselves. At the end of compilation, progend(void) is called to give the back end an opportunity to finalize its
output.

The sample progbeg initializes a register usage mask

\begin{verbatim}
void progbeg(int argc, char *argv[])
{
  rmask = ((~0)<<12)\|1;
}
\end{verbatim}

which is described in Section 5.1. Finalization is unnecessary in the sample,
so progend is an empty macro: \texttt{#define progend(x)}. All back-end functions
except \texttt{address} and \texttt{gen} return nothing. Those that are implemented as macros
instead of as true functions are defined in the configuration file.

\section{3 Symbols}

The front and back ends share two major data structures: symbol table entries
and dag nodes. Dag nodes are described in Section 5. Symbol table entries are
used for variables, constants, and labels. They are represented by pointers to
the following structure.

\begin{verbatim}
typedef struct symbol *Symbol;
struct symbol {
  Xsymbol x;        /* symbol table entries: */
  char *name;       /* type extension for code generator */
  /* name */
}
unsigned char scope;  /* scope level */
unsigned char class;  /* storage class */
unsigned defined:1;  /* 1 if defined */
unsigned temporary:1;  /* 1 if a temporary */
unsigned generated:1;  /* 1 if a generated identifier */
unsigned addressed:1;  /* 1 if its address is taken */
unsigned structarg:1;  /* 1 if parameter is a struct */
Type type;  /* data type */
union {
    int label;  /* labels: label value */
    struct {  /* constants: */
        Value v;  /* value */
        Symbol loc;  /* out-of-line location */
    } c;
    int seg;  /* globals, statics: definition segment */
} u;

The scope, class, and type fields give the symbol’s scope level, its storage class, and its type, respectively. Most of the bit fields flag self-explanatory attributes for each symbol; fields relevant only to the front end are elided above.

Scope values classify symbols as constants, labels, or variables. For labels, constants, and some variables, a field of the union u supplies additional data.

Labels have a scope equal to the enumeration constant LABELS, and u.label is the numeric value of the label. The name of a label is the string representation of u.label. Labels have no type or class.

Constants have a scope equal to CONSTANTS, a class equal to STATIC, and a name equal to the string representation of the constant. The actual value of the constant is stored in the u.c.v field, which is defined by

typedef union value {  /* constant values: */
    char sc;  /* SIGNED/plain CHAR */
    unsigned char uc;  /* UNSIGNED CHAR */
    short ss;  /* SIGNED SHORT */
    unsigned short us;  /* UNSIGNED SHORT */
    int i;  /* INT */
    unsigned int u;  /* UNSIGNED */
    float f;  /* FLOAT */
    double d;  /* DOUBLE */
    char *p;  /* POINTER to anything */
} Value;

If a variable is generated to hold the constant, u.c.loc points to the symbol table entry for that variable.
Variables have a scope equal to GLOBAL, PARAM, or LOCAL+k for nesting level k. class is STATIC, AUTO, EXTERN, or REGISTER. The name of most variables is the name used in the source code. For temporaries and other generated variables, name is a digit sequence. temporary and generated are set for temporaries, and generated is set for labels and other generated variables, e.g., those that hold constants. structarg is described in Section 5.2. For global and static variables, u.seg gives the logical segment in which the variable is defined (see Section 3.3).

The type field for constants and variables points to a structure that describes types. The size and align fields of this structure give, respectively, the size and alignment constraints of the type in bytes.

The x field is an 'extension' in which the back end stores target-specific data for the symbol. The sample has only two fields:

```c
typedef struct {
  char *name;  /* name for back end */
  int offset;  /* frame offset */
} Xsymbol;
```

This definition appears in the configuration file. p->name identifies the symbol to the front end, but the back end may need to emit a different 'name'. For example, the 'name' for locals is usually an offset from a frame pointer. p->x.name is the back end's name for the symbol. For parameters and locals, p->x.offset is the offset from the frame pointer (see Sections 4.1 and 4.2).

### 3.1 defsymbol

Whenever the front end defines a new symbol with scope CONSTANTS, LABELS, or GLOBAL or a static variable, it calls defsymbol(Symbol p) to give the back end an opportunity to initialize its Xsymbol fields. The sample's defsymbol is

```c
void defsymbol(Symbol p) {
  if (p->scope == CONSTANTS)
    p->x.name = p->name;
  else if (p->generated)
    p->x.name = sprintf("L%p", p->name);
  else
    p->x.name = sprintf("_%p", p->name);
}
```

The back end's name for a constant is just the constant itself. For generated symbols including labels, x.name is the front end's name, which is a digit string, prefixed with an L, and an underscore prefixes the names of other symbols. sprintf returns a pointer to a string formatted as specified by its printf-style arguments.
For scope PARAM and LOCAL+∗, the Xsymbol fields are initialized by function (see Section 4.1) and local (see Section 4.2), respectively, and symbols that represent address computations are initialized by address (see Section 4.3).

3.2 import and export

A symbol can be exported or imported. Non-static variables and functions are exported in order to be available to other, separately compiled modules. Likewise, variables and functions used in one module, but defined in another one, are imported in the former module. The front end identifies an exported or imported symbol by calling export(Symbol p) or import(Symbol p). export is always called before the symbol is defined; import, however, may be called any time, before or after the symbol is used.

The sample back end emits an assembler directive for export, but no output is required for import:

#define export(p) print(".globl %s\n", (p)->x.name)
#define import(p)

print is a front-end function similar to the standard printf.

3.3 segment

The front end manages four logical segments: CODE, BSS, DATA, and LIT. Executable code is emitted into the CODE segment, uninitialized variables are defined in the BSS segment, initialized variables are defined and initialized in the DATA segment, and constants appear in the LIT segment.

The front end announces a segment change by calling segment(int s) where s is one of the segments listed above. segment maps the logical segments onto the segments provided by the target machine. The CODE and LIT segments can be mapped to read-only segments; the others must be mapped to read/write segments. The sample mapping is

void segment(int s) {
  switch (s) {
    case CODE: print(".text\n"); break;
    case LIT: print(".text 1\n"); break;
    case DATA:
      case BSS: print(".data\n"); break;
    }
  }

3.4 global

global(Symbol p) emits code to define a global variable. The front end will have already directed the definition to the appropriate logical segment by calling
segment and set p->u.seg to that segment, and it will follow the call to global
with any appropriate calls to the data initialization functions. global handles
the necessary alignment adjustments and the actual definition.

The sample definition for global y is simply y's x.name field preceded by an
alignment directive, if necessary:

```c
void global(Symbol p) {
    switch (p->type->align) {
    case 2: print("\n.align 1; "); break;
    case 4: print("\n.align 2; "); break;
    case 8: print("\n.align 3; "); break;
    }
    printf("%s: ", p->x.name);
}
```

3.5 defconst

defconst(int ty, Value v) emits the scalar v. ty indicates which field of v
is to be emitted according the following table.

<table>
<thead>
<tr>
<th>ty</th>
<th>v field</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>v.uc</td>
<td>character</td>
</tr>
<tr>
<td>S</td>
<td>v.us</td>
<td>short</td>
</tr>
<tr>
<td>I</td>
<td>v.i</td>
<td>int</td>
</tr>
<tr>
<td>U</td>
<td>v.u</td>
<td>unsigned</td>
</tr>
<tr>
<td>P</td>
<td>v.p</td>
<td>any pointer type</td>
</tr>
<tr>
<td>F</td>
<td>v.f</td>
<td>float</td>
</tr>
<tr>
<td>D</td>
<td>v.d</td>
<td>double</td>
</tr>
</tbody>
</table>

The codes S, I, ... are identical to the type suffixes used for the dag operators,
which are described in Section 5. The signed fields v.uc and v.us can be
used instead of v.uc and v.us, but defconst must initialize only the specified
number of bits.

The sample defconst is

```c
void defconst(int ty, Value v) {
    switch (ty) {
    case C: print(".byte %d\n", v.uc); break;
    case S: print(".word %d\n", v.us); break;
    case I: print(".long %d\n", v.i ); break;
    case U: print(".long 0x%lx\n", v.u ); break;
    case P: print(".long 0x%lx\n", v.p ); break;
    case F:
        print(".long 0x%lx\n", ((unsigned *) &v.f)[0]);
        break;
    
```
case D:
    print(".long 0x%x,0x%x\n", ((unsigned *) &v.d)[0],
         ((unsigned *) &v.d)[1]);
    break;
}
}

In the production compilers, defconst accommodates cross-compilation, so it
corrects for different representations and byte orders.

If ty is P, v.p holds a numeric constant of some pointer type. These originate
from declarations like char *p=(char *)0xF0. defaddress emits addresses
relative to a symbol.

Few ANSI C compilers can leave the encoding of floating-point constants to
the assembler, because few assemblers can cope with the effect of casts on these
constants. For example, the correct initialization for
double x = (float)0.3;
is .long 0x999a3f99,0x0. The directive .double 0.3 would erroneously ini-
tialize x to the equivalent of
.long 0x99993f99,0x0999a9999
because it cannot represent the effect of the cast.

3.6 defstring, defaddress, and space

defstring(int len, char *s) emits code to initialize a string of length len
to the characters in s. The front end converts escape sequences, like \n, into
the corresponding ASCII characters.

defaddress(Symbol p) emits the address denoted by p. space(int n)
emits code to allocate n zero bytes.

The sample defstring, defaddress, and space are

void defstring(int len, char *s) {
    while (len-- > 0)
        print(".byte %d\n", *s++);
}

#define defaddress(p) print(".long %s\n", (p)->x.name)

#define space(x) print(".space %d\n", (x))
4 Functions

The front end completely consumes each function before passing any part of the function to the back end. This organization permits certain optimizations. For example, only by processing complete functions can the front end identify the locals and parameters whose address is not taken; only these variables may be assigned to registers.

4.1 function

The front end accumulates functions into private data structures. At the end of each function, it calls function to generate and emit code. The typical form of function is

```c
void function(Symbol f, Symbol caller[], Symbol callee[], int ncalls) {
    ...initialize
    gencode(caller, callee);
    ...emit prologue
    emitcode();
    ...emit epilogue
}
```

gencode is a front-end routine that traverses the front end's private structures and passes each dag to the back end's gen (see Section 5.1), which selects code, annotates the dag to record its selection, and returns a dag pointer. emitcode is a front-end routine that traverses the private structures again and passes each of the pointers from gen to emit (see Section 5.2), which emits the code.

This organization offers the back end flexibility in generating function prologue and epilogue code. Before calling gencode, function initializes the xsymbol fields of the function's parameters, as described below, and does other per-function initializations, if necessary. After calling gencode, the size of the activation record, or frame, the number of registers used, etc. are known; this information is usually needed to emit the prologue. After calling emitcode to emit the code for the body of the function, function emits the epilogue.

The argument f to function is the pointer to the symbol table entry for the current function, and ncalls is the number of calls to other functions made by the current function. ncalls is useful on targets like the SPARC where 'leaf' functions get special treatment.

caller and callee are arrays of pointers to symbol table entries; each is terminated with a zero pointer. The symbols in caller are the function parameters as passed by a caller; those in callee are the parameters as seen within the function. For most functions, the symbols in each array are the same, but they can differ in both class and type. For example, in
foo(x) float x; { ... }

A call to foo passes the actual argument as a double. Within foo, x is a float. Thus, caller[0]->type refers to ‘double’ and callee[0]->type refers to ‘float.’ And in

```
int strlen(register char *s) { ... }
```

caller[0]->class is AUTO and callee[0]->class is REGISTER. Even without register declarations, the front end assigns frequently referenced parameters to the REGISTER class, and callee’s class is set accordingly. This assignment is made only when there are no explicit register locals to avoid interfering with the programmer’s intentions (see Section 4.2).

caller and callee are passed to gencode. If caller[i]->type is not equal to callee[i]->type or if caller[i]->class is not equal to callee[i]->class, gencode generates an assignment of caller[i] to callee[i]. If the types are not equal, this assignment may include a conversion; for example, the assignment to x in foo includes a truncation of a double to a float. For parameters that include register declarations, function must assign a register and initialize the x field accordingly, or change the callee’s class to AUTO.

Function could also change callee[i]->class from AUTO to REGISTER if it wished to assign a register to that parameter. On the MIPS, for example, some of the parameters are passed in registers, so function assigns those registers to the corresponding callees in leaf functions. If, however, callee[i]->addressed is set, the address of the parameter is taken in the function body, and it must be stored in memory on most machines.

Initialization of the Xsymbol fields of the symbols in caller and callee depends on the frame layout, which is target specific. Figure 1 shows the layout of the VAX frame. The stack grows towards lower addresses and towards the top of the page.

Arguments are referenced by displacement-mode addressing with positive offsets from register ap, so the first argument is at address 4(ap). Locals are referenced via negative offsets from fp, e.g., the first local is at -4(fp). The ‘argument build area’ is used to store arguments to functions that are called by the current function. The front end ‘un-nests’ calls so that the back end does not need to deal with nested calls. The argument build area can thus be used for all calls and must be large enough to hold the largest argument list. When a function is called, the caller’s argument build area becomes the callee’s ‘actual arguments’.

Typical VAX calling sequences can handle nested calls, so using an argument build area is not strictly necessary. But other targets, such as the MIPS, require this approach, so it’s used here to illustrate the technique. This approach also has the advantage that stack overflow can occur only at function entry, which is a useful on targets that require explicit prologue code to detect stack overflow.

The sample version of function is
static int framesize;         /* size of activation record */
static int offset;            /* current frame offset */
static int argbuildsize;      /* size of argument build area */

void function(Symbol f, Symbol caller[],
               Symbol callee[], int ncalls) {
int i;

    offset = 4;
    for (i = 0; caller[i] && callee[i]; i++) {
        offset = roundup(offset, caller[i]->type->align);
        callee[i]->x.offset = callee[i]->x.offset = offset;
        callee[i]->x.name = callee[i]->x.name = stringf("%d(ap)", offset);
        offset += callee[i]->type->size;
        callee[i]->class = AUTO;
    }
    usedmask = argbuildsize = framesize = offset = 0;
genode(caller, callee);
    printf("%s: .word 0x%x\n", f->x.name, usedmask&0x3f);
    framesize += 4*nregs + argbuildsize;
    printf("subl2 $%d, sp\n", framesize);
    if (isstruct(fretur(f->type)))
}


```c
print("movl r1,-4(fp)\n");
emitcode();
if (glevel > 1)
    print("ret\n");
}
```

The VAX calls instruction saves the general registers specified by the entry mask, ep, fp, and the return address, pc, as shown in the frame figure above. function computes the size of the locals and argument build area, given by framesize and argbuildsize, respectively.

The first part of function initializes the x.offset and x.name fields of each caller and callee symbol to the appropriate offset and name, respectively. The running offset is rounded up to the alignment for each argument. roundup(n,m) is a front-end macro that returns n rounded up to the next multiple of m. Depending on the type metrics, the size of an argument may not be a multiple of longwords (e.g., 3-byte structures), but the front end ensures that the minimum alignment for each caller[i] is that for integers, which keeps the VAX stack longword-aligned. stringd(n) is a front-end function that returns the string representation of the integer n.

This version of function does not support register declarations, so each callee's storage class is set to AUTO.

During code generation, argbuildsize is increased when code for calls is generated (see Section 5.1), offset is adjusted in response to the definition of locals and block boundaries, and framesize records offset's maximum (see Sections 4.2 and 4.4). Before calling gencode, function clears usedmask, which records the registers used in the function body, and clears argbuildsize, framesize, and offset.

After gencode returns, usedmask and framesize hold the information needed to generate the prologue. framesize is adjusted to include the argument build area and space for saving all of the registers. This space, which is in addition to that specified by the register save mask, is used to save registers across those instructions that destroy fixed registers, e.g., movc3. The subl2 instruction allocates the remainder of the frame.

The last instruction of the prologue is emitted only for functions that return structures. For a function type ty, freturn(ty) gives the type of the value returned by the function, and isstruct(ty) is true if ty is a structure or union type. Section 5 gives details on returning structures.

If JUMP_ON_RETURN had been defined as 1 in the configuration, returns would have been followed by a jump to the label that follows the code emitted by emitcode. If the front end is passed the -gn option, it sets the global variable glevel to n and behaves as if JUMP_ON_RETURN is 1, so the code above emits the necessary ret instruction at the end of each function.
4.2 local

During the execution of gencode, the front end announces local variables by calling local(Symbol p), where p points to the relevant symbol table entry. It announces temporaries likewise; these have p->temporary set. local must initialize p's Symbol fields. That is, it must set p->x.offset and p->x.name so that they identify a stack offset or register number, depending on the availability of registers and on the value of p->class, which is AUTO or REGISTER.

For each block, the front end first announces locals with explicit register declarations, in order of declaration, to permit programmer control of register assignment. Then it announces the rest, starting with those that appear to be most frequently referenced. It assigns REGISTER class to even these locals if their address is never taken and if their estimated frequency of use exceeds two. This announcement order and class override collaborate to put the most promising locals in registers even if no registers were declared. As with parameters, local could assign a register to p and change p->class from AUTO to REGISTER, but it should do so only if p->addressed is not set.

If p->class is REGISTER, local can decline to allocate a register by setting p->class to AUTO and initializing p->x.offset and p->x.name to the appropriate frame offset and address string, respectively. This choice is illustrated by the sample version:

```c
void local(Symbol p) {
    offset = roundup(offset + p->type->size, p->type->align);
    offset = roundup(offset, 4);
    p->x.offset = -offset;
    p->x.name = sprintf("%d(fp)", -offset);
    p->class = AUTO;
}
```

The second roundup keeps offset and hence the stack aligned on longwords, as described above.

4.3 address

The front end calls address(Symbol q, Symbol p, int n) to initialize q->x to a symbol that represents an address of the form x+n, where x is the address represented by p and n is positive or negative. Like defsymbol, address initializes q->x, but does so based on the values of p->x and n. The sample address is

```c
void address(Symbol q, Symbol p, int n) {
    if (p->scope == GLOBAL || p->class == STATIC || p->class == EXTERN)
        q->x.name = sprintf("%s%+%s%d", p->x.name, n >= 0 ? "+" : "", n);
    else {
```
q->x.offset = p->x.offset + n;
q->x.name = sprintf("%d(%s)", q->x.offset,
              p->scope == PARAM ? "ap" : "fp");
}

which computes q->x.offset and q->x.name for locals and parameters, or sets q->x.name to p->x.name concatenated with +n or -n for other variables.

For example, in

```c
struct node { struct node *link; int count; } a;
f() { int b[10]; b[4] = a.count; ... }
```
suppose a and b point to the symbol table entries for a and b, respectively. a->x.name is set to "_a" by defsymbol, and b->x.offset and b->x.name are set to, respectively, -40 "-40(fp)" by local. address(q1,a,4) is called with q1 representing the address of a.count, and q1->x.name is set to "_a+4". Likewise, address(q2,b,16) sets q2->x.offset and q2->x.name to, respectively, -24 and "-24(fp)", which together denote the address of b[4].

address accepts globals, parameters, and locals.

### 4.4 blockbeg and blockend

Source-language blocks bracket the lifetime of locals. gencode announces the beginning and end of a block by calling blockbeg(Env *e) and blockend(Env *e), respectively. Env is target specific and typically includes the data necessary to reuse that portion of the local frame space associated with the block and to release any registers assigned to locals within the block. The sample Env is

```c
typedef struct {
    unsigned rmask;
    int offset;
} Env;
```

The fields save the values of rmask and offset at the beginning of a block so that they can be restored on the end of the block. The sample blockbeg and blockend are thus

```c
void blockbeg(Env *e) {
    e->rmask = rmask;
    e->offset = offset;
}

void blockend(Env *e) {
    if (offset > framesize)
        framesize = offset;
    offset = e->offset;
    rmask = e->rmask;
}
```

blockend also updates framesize if the locals for the current block require more space than previous blocks. The sample could do without the rmask field,
but if its local assigned registers to locals, it would need the field to release those registers.

Temporaries — locals with temporary set — to which local assigned registers live only for the expressions in which they are used. They are announced by local as usual, but are used only in the dags passed to next call on gen (see 5.1). gen can thus release all registers assigned to temporaries.

5 Dags

Executable code is specified by dags. A function body is a sequence of forests of dags, each of which is passed the back end via gen, as described below. Dag nodes, or simply nodes, are defined by

```c
typedef struct node *Node;

struct node {
    /* dag nodes: */
    Opcode op;  /* operator */
    short count;  /* reference count */
    Symbol sym[MAXSYMS];  /* symbols */
    Node kids[MAXKIDS];  /* operands */
    Node link;  /* next dag in the forest */
    Xnode x;  /* back-end's type extension */
};
```

The kids point to the operand nodes. Some operators also take symbol table pointers as operands; these appear in the sym array. The default and minimum allowable value for both MAXKIDS and MAXSYMS is 2; larger values can be defined for the back end’s convenience in the configuration file. count holds the number of references to this node from kids in other nodes. link points to the root of the next dag in the forest.

The x field is the back end’s ‘extension’ to nodes. The configuration defines the type Xnode to hold the per-node data that the back end needs to generate code. The sample Xnode is

```c
typedef struct {
    unsigned visited:1;  /* 1 if dag has been linearized */
    int reg;  /* register number */
    unsigned xmask;  /* unshifted register mask */
    unsigned busy;  /* busy regs */
    int argoffset;  /* ARG: argument offset */
    Node next;  /* next node on emit list */
} Xnode;
```

Section 5.1 describes the fields.

The op field holds an operator. The last character of each is a type suffix from Table 1. For example, the generic operator ADD has the variants ADDI, ADDU, ADDP, ADDF, and ADDD.
<table>
<thead>
<tr>
<th>type suffix</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>char</td>
</tr>
<tr>
<td>S</td>
<td>short</td>
</tr>
<tr>
<td>I</td>
<td>int</td>
</tr>
<tr>
<td>U</td>
<td>unsigned</td>
</tr>
<tr>
<td>P</td>
<td>any pointer type</td>
</tr>
<tr>
<td>F</td>
<td>float</td>
</tr>
<tr>
<td>D</td>
<td>double</td>
</tr>
<tr>
<td>B</td>
<td>structure or block</td>
</tr>
<tr>
<td>V</td>
<td>void</td>
</tr>
</tbody>
</table>

Table 1: Type Suffixes.

Table 2 lists each generic operator, its valid type suffixes, and the number of kids and syms that it uses; multiple values for kids indicate type-specific variations, which are detailed below. For most operators, the type suffix denotes the type of operation to perform and the type of the result. Exceptions are ADDP, in which the first integer operand is added to the second pointer operand, and SUBP, which subtracts the second integer operand from the first pointer operand.

The operators for assignment, comparison, arguments, and some calls return no results; their type suffixes denote the type of operation to perform.

The leaf operators yield the address of a variable or the value of a constant. syms[0] identifies the variable or constant.

The unary operators accept and yield a number, except for INDIR, which accepts an address and yields the value at that address. There is no BCOMI; signed integers are complemented using BCOMU. The binary operators accept two numbers and yield one.

The type suffix for a conversion operator denotes the type of the result. For example, CVUI converts an unsigned (U) to an signed integer (I). Conversions between unsigned and short and between unsigned and character are unsigned conversions; those between integer and short and between integer and character are signed conversions. For example, CVSU converts an unsigned short to an unsigned while CVS1 converts an signed short to a signed integer.

The front end composes conversions to form those not in the table. For example, it converts a short to a float by first converting it to an int and then a double. The 16 conversion operators are represented by arrows in Figure 2. Composed conversions follow the path from the source type to the destination type.

There is no CVUD; conversion of an unsigned u to a double is done by the equivalent of the expression

\[(\text{int})u >= 0 \iff (\text{double})(\text{int})u : (\text{double})(\text{int})u + \text{UINT}_\text{MAX} + 1\]
<table>
<thead>
<tr>
<th>syms</th>
<th>kids</th>
<th>operator</th>
<th>type suffixes</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>ADDRF</td>
<td>P</td>
<td>address of a parameter</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ADDRG</td>
<td>P</td>
<td>address of a global</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ADDRL</td>
<td>P</td>
<td>address of a local</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>CNST</td>
<td>CSIUPFD</td>
<td>constant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>BCOM</td>
<td>U</td>
<td>bitwise complement</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>CVC</td>
<td>IU</td>
<td>convert from char</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>CVD</td>
<td>I F</td>
<td>convert from double</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>CVF</td>
<td>D</td>
<td>convert from float</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>CVI</td>
<td>CS U D</td>
<td>convert from int</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>CVP</td>
<td>U</td>
<td>convert from pointer</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>CVS</td>
<td>IU</td>
<td>convert from short</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>CVU</td>
<td>CSI P</td>
<td>convert from unsigned</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>INDIR</td>
<td>CSI PFDB</td>
<td>fetch</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>NEG</td>
<td>I FD</td>
<td>negation</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>ADD</td>
<td>IUPFD</td>
<td>addition</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>BAND</td>
<td>U</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>BOR</td>
<td>U</td>
<td>bitwise inclusive OR</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>BXOR</td>
<td>U</td>
<td>bitwise exclusive OR</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>DIV</td>
<td>IU FD</td>
<td>division</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>LSH</td>
<td>IU</td>
<td>left shift</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>MOD</td>
<td>IU</td>
<td>modulus</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>MUL</td>
<td>IU FD</td>
<td>multiplication</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>RSH</td>
<td>IU</td>
<td>right shift</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>SUB</td>
<td>IUPFD</td>
<td>subtraction</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>ASGN</td>
<td>CSI PFDB</td>
<td>assignment</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>EQ</td>
<td>IU FD</td>
<td>jump if equal</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>GE</td>
<td>IU FD</td>
<td>jump if greater than or equal</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>GT</td>
<td>IU FD</td>
<td>jump if greater than</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>LE</td>
<td>IU FD</td>
<td>jump if less than or equal</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>LT</td>
<td>IU FD</td>
<td>jump if less than</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>NE</td>
<td>IU FD</td>
<td>jump if not equal</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>ARG</td>
<td>I PFDB</td>
<td>argument</td>
</tr>
<tr>
<td>0  1  2</td>
<td>CALL</td>
<td>I FDBV</td>
<td>function call</td>
<td></td>
</tr>
<tr>
<td>0  1</td>
<td>RET</td>
<td>I FD V</td>
<td>return from function</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>JUMP</td>
<td>V</td>
<td>unconditional jump</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>LABEL</td>
<td>V</td>
<td>label definition</td>
</tr>
</tbody>
</table>

Table 2: Node Operators.
where `UINT_MAX` is the ANSI-specified maximum value for an unsigned. Likewise, there is no `CVDU`; conversion of a double `d` to an unsigned is done by the equivalent of the expression

\[
d \geq INT_MAX + 1 \ ? \ \text{(unsigned)}(\text{int})\left(d-(INT_MAX+1)\right) + \text{INT_MAX} + 1 \\
: \ \text{(unsigned)}(\text{int})d
\]

where `INT_MAX` is the ANSI-specified maximum value for a signed integer.

`ASGN` stores the value of `kids[1]` in the cell addressed by `kids[0]`. `syms[0]` and `syms[1]` point to symbol table entries for integer constants that give the size of the value and its alignment, respectively. These are most useful for `ASGNB`, which implements structure assignment and other 'block moves' (e.g., initialization of automatic arrays).

For the comparisons, `syms[0]` points to a symbol table entry for the label to jump to if the comparison is true. `JUMPV` is an unconditional jump to the address computed by `kids[0]`. `LABEL` defines the label given by `syms[0]` and is otherwise a no-op.

Function calls have a `CALL` node preceded by zero or more `ARG` nodes. The front end 'un-nests' function calls, so `ARG` nodes are always associated with the next `CALL` node in the forest. The order of the `ARG` nodes is right-to-left unless the configuration parameter `LEFT_TO_RIGHT` is defined, which it is for the sample.

`ARG` nodes establish the value computed by `kids[0]` as the next argument. `syms[0]` and `syms[1]` point to symbol table entries for integer constants that give the size of the argument and its alignment, respectively.

In `CALL` nodes, `kids[0]` computes the address of the callee. `CALLB` calls functions that return structures; `kids[1]` computes the address of a temporary local variable to hold the returned value. There is no `RETB`; the front end uses a `RETV` preceded by an `ASGNB` to the structure addressed by the first local. The `CALLB` code and the function prologue must collaborate to store the `CALLB`'s `kids[1]` into the callee's first local. `function` (Section 4.1), `local` (Section 4.2) and the code emitted below for `CALLB` (Section 5.2) illustrate such collaboration. `CALLB` nodes have a `count` of 0 because the front end references the temporary wherever the returned value is referenced.
In RET nodes, \texttt{kids[0]} computes the value returned, except for RETV nodes, which are childless.

Character and short integer arguments are always promoted to the corresponding integer type even in the presence of a prototype. The promoted values are converted back to the intended type upon entry to the function. The front end accomplishes this conversion by specifying the intended types for the callee as described in Section 4.1. For example, the body for

\[
f(\text{char } c) \{ f(c); \}
\]

becomes two forests, which are linearized below; the \texttt{sym}s columns list the \texttt{x.name} fields.

\begin{tabular}{llll}
\texttt{# op} & \texttt{count} & \texttt{kids} & \texttt{sym}s \\
1. & ADDRFP & 1 & 4(ap) \\
2. & ADDRFP & 1 & 4(ap) \\
3. & INDIRI & 1 & 2 \\
4. & CVIC & 1 & 3 \\
5. & ASGN & 0 & 1, 4 \\
\hline
1. & ADDRFP & 1 & 4(ap) \\
2. & INDIRC & 1 & 1 \\
3. & CVCI & 1 & 2 \\
4. & ARG1 & 0 & 3, 4 \\
5. & ADDRGP & 1 & _f \\
6. & CALL & 0 & 5, 4 \\
\end{tabular}

The first forest holds one dag, which converts the actual argument to the intended type. The second forest holds two dags. The first (nodes 1–4) promotes \texttt{c} to pass it as an integer, and the second (nodes 5 & 6) calls \texttt{f}.

Unsigned variants of \texttt{ASGN}, \texttt{INDIR}, \texttt{ARG}, \texttt{CALL}, and \texttt{RET} were omitted as unnecessary. Signed and unsigned integers have the same size, so the corresponding signed operator is used instead. Likewise, there is no \texttt{CALLP} or \texttt{RETP}. A pointer is returned by using \texttt{CVPU} and \texttt{RETI}. A pointer-valued function is called by using \texttt{CALLI} and \texttt{CVUP}.

In Table 2, the operators listed at and following \texttt{ASGN} are used for their side-effects. They always appear as roots in the forest of dags, and they appear in the order in which they must be executed. Except for \texttt{CALLD}, \texttt{CALLF} and \texttt{CALLI}, their reference counts are always zero. Even these \texttt{CALL} nodes have zero reference counts when their values are unused.

### 5.1 \texttt{gen}

The front end calls \texttt{gen} to select code. It passes \texttt{gen} a forest of dags. For example,
int i, *p; f() { i = *p++; }

yields the forest linearized below.

<table>
<thead>
<tr>
<th>#</th>
<th>op</th>
<th>count</th>
<th>kids</th>
<th>syms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADDRGP</td>
<td>2</td>
<td>-P</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>INDIRP</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>CNSTI</td>
<td>1</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>ADDP</td>
<td>1</td>
<td>2 3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ASGNP</td>
<td>0</td>
<td>1 4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ADDRGP</td>
<td>1</td>
<td></td>
<td>_i</td>
</tr>
<tr>
<td>7</td>
<td>INDIRI</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ASGNI</td>
<td>0</td>
<td>6 7</td>
<td></td>
</tr>
</tbody>
</table>

This forest consists of three dags, rooted at nodes 2, 5, and 8 above. The INDIRP node, which fetches the value of p, comes before node 5, which changes p, so the original value of p is available for subsequent use by node 7, which fetches the integer pointed to by that value.

`gen` traverses the forest and selects code, but it emits nothing because it may be necessary to determine, for example, the registers needed before the function prologue can be emitted (see Section 4.1). So `gen` merely annotates the nodes to identify the code selected, and it returns a pointer that is ultimately passed to the back end's `emit` to actually output the code. Once the front end calls `gen`, it does not inspect the contents of the nodes again, so `gen` may modify them freely.

The sample code generator emits naive code, so `gen` concerns itself mainly with register allocation. The sample `gen`

```c
Node gen(Node p) {
    Node head, *last;

    for (last = &head; p = p->link)
        last = linearize(p, last, 0);
    for (p = head; p = p->x.next) {
        alloc(p);
        if (p->count == 0 && sets(p))
            putreg(p);
    }
    return head;
}
```

linearizes each dag in the forest, in execution order, and then allocates registers for each node. If the node sets a register, but no subsequent node references it, which is indicated by a reference `count` of 0, `gen` releases the register. Finally, it returns the linearized node list for traversal by `emit`.

`gen` linearizes the dags before allocating registers to simplify the insertion of spills and reloads. The function
static Node *linearize(Node p, Node *last, Node next) {
    if (p && !p->x.visited) {
        last = linearize(p->kids[0], last, 0);
        last = linearize(p->kids[1], last, 0);
        p->x.visited = 1;
        *last = p;
        last = &p->x.next;
    }
    *last = next;
    return last;
}

linearizes the dag at p and inserts it into the growing list of nodes between
*last and next.

ralloc handles register allocation for a single node.

static void ralloc(Node p) {
    int i;

    switch (generic(p->op)) {
    case ARG:
        argoffset = roundup(argoffset, p->sym[s] u.c.v.i);
        p->x.argoffset = argoffset;
        argoffset += p->sym[s] u.c.v.i;
        if (argoffset > argbuildsize)
            argbuildsize = roundup(argoffset, 4);
        break;
    case CALL:
        argoffset = 0;
        break;
    }
    for (i = 0; i < MAXKIDS; i++)
        putreg(p->kids[i]);
    p->x.busy = rmask;
    if (needsreg(p))
        getreg(p);
}

generic strips the type suffix from an operator and thus simplifies the switch.
ralloc calls putreg to release the registers allocated for the node's children and
then it calls getreg to allocate a register for the node itself if it needs one. It
sets the node's x.busy to record the busy registers; emit needs this information
for a few operators. The register state is encoded in rmask; rmask & (1<<r) is 1
if register r is busy, for r from 0 to 11.

CALL and ARG nodes require extra steps. ARG nodes produce no result; instead,
they store the value computed by kids[0] into the next location in the
argument build area. `syms[0]` and `syms[1]` point to constant symbols that give the size and alignment of the argument. `argoffset` is the running offset into the argument build area. `argoffset` is rounded up to the appropriate alignment boundary, saved in the node's `x.argoffset` for use in `emit`, and incremented by the size of the argument. `argbuildsize` tracks the maximum `argoffset` needed by the current function. The case for CALL nodes clears `argoffset` for the next set of ARG nodes.

`getreg` accepts a node and allocates a register for it:

```c
static void getreg(Node p) {
    int r, m = optype(p->op) == D ? 3 : 1;

    for (r = 0; r < nregs; r++)
        if (((rmask&({m<r})) == 0) {
            p->x.rmask = m;
            p->x.reg = r;
            rmask |= sets(p);
            usedmask |= sets(p);
            return;
        }
    r = spillee(p, m);
    spill(r, m, p);
    getreg(p);
}
```

`optype(op)` returns the type suffix of operator `op`. `m` is set to the mask 1 if the result of `p->op` needs an ordinary register and 3 if it needs a double register. `getreg` loops over the registers. If it finds one that's free, it sets the node's `x.reg` field to the register allocated and the `x.rmask` field to the mask. It also updates `usedmask`, which is used to generate the prologue described in Section 4.1; `sets(p)` returns `p->x.rmask<<p->x.reg`. If no registers are free, `getreg` spils a register and calls itself recursively to try again. Section 6 describes spills.

`putreg` releases registers:

```c
static void putreg(Node p) {
    if (p && --p->count <= 0)
        rmask &= ~sets(p);
}
```

Dags can use result registers multiple times, so `putreg` decrements the reference count and frees the register only when the last reference is removed by clearing the appropriate bits in `rmask`. 
5.2 emit

emit emits the linearized forest. The sample walks down the list, switches on the opcode to identify the code to emit:

```c
c void emit(Node p) {
    for (; p; p = p->x.next) {
        Node a = p->kids[0], b = p->kids[1];
        int r = p->x.reg;
        switch (p->op) {
        case CNSTC: case CNSTI: case CNSTP:
            case CNSTS: case CNSTU:
                printf("movl \%s,\%d\n", p->syms[0]->x.name, r);
                break;
        case ADDRGP: case ADDRFP: case ADDRLP:
            printf("moval \%s,\%d\n", p->syms[0]->x.name, r);
            break;
        ...
        }
    }
}
```

The individual cases emit naive code for a single operator. The CNST cases above emit a VAX instruction that loads a constant into a register. defsymbol stored the constant string in p->syms[0]->x.name, and ralloc stored the result register name p->x.reg. The ADDR cases above emit a VAX instruction that moves the address of a variable into the result register.

Most of the unary operators share a common pattern, which the sample abstracts into a macro:

```c
#define suffix(p) ".f8wlll1l.l"[o ctype((p)->op)]
#define unary(inst) print("%s\%c \%d,\%d\n", inst, suffix(p), a->x.reg, r)
```

```c
case BCOMU: unary("mcom" ); break;
    case NEGD: case NEGF: case NEGI: unary("mneg" ); break;
    case CVCI: unary("cvth" ); break;
    case CVCU: unary("movzb" ); break;
    case CVSI: unary("cvtv" ); break;
    case CVSU: unary("movzw" ); break;
    case CVDF: case CVDI: unary("cvttd" ); break;
    case CVFD: unary("cvtf" ); break;
    case CVUC: case CVUS: unary("cvtl" ); break;
    case CVIC: case CVIS: case CVID: unary("cvtl" ); break;
    case CVIU: case CVUI: unary("mov" ); break;
    case CVPU: case CVUP: unary("mov" ); break;
```
unary emits the VAX operator, the VAX type suffix — which is computed by indexing a string of such suffixes — and the source and destination registers. Most of the binary operators

```c
#define binary(inst) print("%s%c3 r%\d,r%\d,r%\d\n", inst, suffix(p), \
    b->x.reg, a->x.reg, r)

  case BANDU:    binary("bic"); break;
  case BORU:     binary("bis"); break;
  case BXORU:    binary("xor"); break;
  case ADDD:     case ADDF:     binary("add"); break;
  case ADDI:     case ADDP:     case ADDU:    binary("add"); break;
  case SUBD:     case SUBF:     case SUBU:    binary("sub"); break;
  case SUBL:     case SUBP:     case SUBU:    binary("sub"); break;
  case MULD:     case MULF:     case MULU:    binary("mul"); break;
  case MULI:     case MULP:     case MULU:    binary("mul"); break;
  case DIVD:     case DIVF:     case DIVI:    binary("div"); break;

and all of the comparisons

```c
#define compare(cp) print("cmp%c r%\d,r%\d; j%\s %\s\n", suffix(p), \
    a->x.reg, b->x.reg, cp, p->sym[0]->x.name)

  case EQD:      case EQF:      case EQI:    compare("eql"); break;
  case EQU:      compare("eqlu"); break;
  case GED:      case GEF:      case GEI:    compare("geq"); break;
  case GEU:      compare("gequ"); break;
  case GTD:      case GTF:      case GTI:    compare("gtz"); break;
  case GTU:      compare("gtru"); break;
  case LED:      case LEF:      case LEI:    compare("leq"); break;
  case LEU:      compare("lequ"); break;
  case LTD:      case LTF:      case LTI:    compare("lss"); break;
  case LTU:      compare("lssu"); break;
  case NED:      case NEF:      case NEI:    compare("neq"); break;
  case NEU:      compare("nequ"); break;

are handled similarly.

The cases

```c
  case INDIRC:  case INDIRD:  case INDIRF:  case INDIRI:
  case INDIRP:  case INDIRS:
      print("mov%c (r%\d),r%\d\n", suffix(p), a->x.reg, r);
      break;
  case ASGNC:   case ASGND:   case ASGNF:   case ASGNI:  case ASGNP:  case ASGNS:
      print("mov%c r%\d,(r%\d)\n", suffix(p), b->x.reg, a->x.reg);
      break;
  case JUMPV:
      print("jmp (r%\d)\n", a->x.reg);

```
break;
case LABELV:
    print("%s:", p->sym[0]->x.name);
    break;

emit code to indirectly load and store memory cells, to jump indirectly, and to emit a label definition.
The cases

case ARGD: case ARGF: case ARG1: case ARG2:
    print("mov%cr%d,%d(sp)\n", suffix(p),
        a->x.reg, p->x.argoffset);
    break;
case CALLD: case CALLF: case CALL1: case CALL2:
    save(p->x.busy&0x3e);
    print("calls $0,(r%d)\n", a->x.reg);
    if (p->op != CALLV)
        print("mov%cr0,r%d\n", suffix(p), r);
    restore(p->x.busy&0x3e);
    break;
case RETD: case RETF: case RET1:
    print("mov%cr%d,r0; ret\n", suffix(p), a->x.reg);
    break;
case RETV:
    print("ret\n");
    break;

emit code to move an argument onto the stack, to call a procedure, and to return a value. The ARG cases use x.argoffset, into which malloc stored the stack offset for the argument. Procedures may destroy registers 1–5, so the CALL cases use

static void save(unsigned mask) {
    int i;

    for (i = 1; i < nregs; i++)
        if (mask&(i<i))
            print("movl r%d,%d(fp)\n", i,
                4*i - framesize + argbuildsize);
}

to emit code to save any of those registers that are busy; restore is similar.
The RET cases merely copy any return value into register 0 and return.

Several binary operators require special handling. The shift instructions use a syntax that differs slightly from the other binary instructions:
case RSHI: case LSHI: case LSHU:
  print("ashl r%d,r%d,r%d\n", b->x.reg, a->x.reg, r);
  break;

RSHI and LSHI generate the same code because the front end negates the shift count for RSHI if the configuration parameter VAX is defined. No instructions directly implement unsigned right shift, division, or modulus, so emit uses a field-extraction instruction for the first and library calls for the others:

case RSHU:
  print("subl3 r%d,$32,r0; extzv r%d,r0,r%d,r%d\n", b->x.reg, a->x.reg, r);
  break;

case DIVU:
  save(p->x.busy&0x3e);
  print("pushl r%d; pushl r%d; calls $2,udiv; movl r0,r%d\n", b->x.reg, a->x.reg, r);
  restore(p->x.busy&0x3e);
  break;

case MODU:
  save(p->x.busy&0x3e);
  print("pushl r%d; pushl r%d; calls $2,urem; movl r0,r%d\n", b->x.reg, a->x.reg, r);
  restore(p->x.busy&0x3e);
  break;

Finally, ANSI C requires that a%b equal a-(a/b)*b, so MODI computes it just this way:

case MODI:
  print("divl3 r%d,r%d,r0; mull2 r%d,r0; subl3 r0,r%d,r%d\n", b->x.reg, a->x.reg, r);
  break;

Only the structure instructions remain:

case INDIRB:
  print("moval (r%d),r%d\n", a->x.reg, r);
  break;

case ASCNB:
  save(p->x.busy&0x3f);
  print("mova% $%s,(r%d)\n", p->syms[0]->x.name, b->x.reg, a->x.reg);
  restore(p->x.busy&0x3f);
  break;

case ARGB:
save(p->x.busy&0x3f);
print("movc3 %s,(r%d),%d(sp)\n", p->syms[0]->x.name,
        a->x.reg, p->x.argoffset);
restore(p->x.busy&0x3f);
break;

The scalar INDIRs and ASGNs load and store values directly, but structures won't
fit in registers, so their instructions manipulate addresses instead. ASGNB uses a
block move instruction, which needs the size from p->syms[0]->x.name; it also
destroyes registers 0-5, so emit arranges to save and restore their values. ARG
operates similarly, but copies the structure into the stack instead. Finally,

case CALLB:
    save(p->x.busy&0x3e);
    if (a->x.reg == 1) {
        print("movl r1,r0\n");
        a->x.reg = 0;
    }
    if (b->x.reg != 1)
        print("movl r%d,r1\n", b->x.reg);
    print("calls $0,(r%d)\n", a->x.reg);
    restore(p->x.busy&0x3e);
    break;

is like an ordinary call, but it also passes the address at which to store the return
value in register 1; if the address of the function is already in register 1, it is
first moved out of the way into register 0, which is not otherwise allocated.

If NOARGB is defined (e.g., in the configuration file), the front end uses Sun's
convention for passing structures by value. It builds dags that copy structure
arguments to temporaries, passes pointers to these temporaries, adds an extra
indirection to references to these parameters in the callee, and changes the types
of the callee's formals to reflect this convention. It also sets structarg for these
parameters.

5.3 asmcode

asm("...") directives in C source programs cause emitcode to call asmcode:

void asmcode(char *str, Symbol argv[]) {
    for (; *str; str++)
            print("%s", argv[***str]->x.name);
        else
            print("%c", *str);
    print("\n");
}
str points to the string given in the asm directive, and argv is an array of symbol table pointers.

The front end replaces occurrences of %x that appear in the asm string with %k, where k is a 1-byte integer from 0 to 9. argv[k] is the symbol table entry for identifier x. asmcode emits str, replacing %k with appropriate back-end names. For example, the asm in

```
int x; f(register i) { int j; asm("addl3 %i,%j,%x"); ... }
```

becomes addl3 4(ap),-4(fp),_x.

6 Spills

When getreg finds that all registers are busy, one or more must be spilled now and reloaded when the values are needed again. Handling spills correctly is difficult and a common source of compiler bugs. Test cases that expose bugs in the spill code are necessarily complex. As such, and for completeness, the implementation described in this section is adapted from that used in the production versions of lcc. Reference [6] describes the important differences and explains the rationale behind this spiller.

The dag constructed by the front end minimizes the reevaluation of common subexpressions. Spills are, in a sense, a result of the front end’s eagerness to avoid reevaluation, and handling spills amounts to ‘breaking the dags’ generated by the front end. A node representing a common subexpression is changed so that it stores the value in a temporary, and subsequent references to that node are edited to load the value from the temporary.

Spilling involves three major steps: Identifying the registers to spill, generating the code for the spills at the correct location the output stream, and generating the code for the reloads, again at the correct locations. These steps are performed at the end of getreg:

```
static void getreg(Node p) {
    int r, m = optype(p->op) == D ? 3 : 1;
    ...
    r = spillee(p, m);
    spill(r, m, p);
    getreg(p);
}
```

spillee identifies the register (m==1) or register pair (m==3) to be spilled on behalf of p, and spill generates the spill and reload code. Once r has been spilled, it is available, and the call to getreg is guaranteed to succeed.

ralloc frees registers as soon as possible. If the available registers are exhausted, it is because there are multiple references to the nodes holding the registers, which arise from common subexpressions and from multiple assignment,
augmented assignment, and the operators ++ and --. Consider the following program.

double a[10], b[10];
ing i;
f(i) { i = (a[i]+b[i])*(a[i]-b[i]); }

The initial linearized forest is shown to the left of the vertical line in the display below.

<table>
<thead>
<tr>
<th>#</th>
<th>op</th>
<th>kids</th>
<th>sym</th>
<th>count</th>
<th>count</th>
<th>uses</th>
<th>sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADDRGP</td>
<td>.i</td>
<td>2</td>
<td></td>
<td></td>
<td>r1</td>
<td>r2</td>
</tr>
<tr>
<td>2</td>
<td>INDIRI</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>r2</td>
<td>r3</td>
</tr>
<tr>
<td>3</td>
<td>CNSTI</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>r3</td>
</tr>
<tr>
<td>4</td>
<td>LSHI</td>
<td>2 3</td>
<td>2</td>
<td></td>
<td></td>
<td>r2</td>
<td>r3</td>
</tr>
<tr>
<td>5</td>
<td>ADDRGP</td>
<td>.a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>r3</td>
</tr>
<tr>
<td>6</td>
<td>ADDP</td>
<td>4 5</td>
<td>1</td>
<td></td>
<td></td>
<td>r2</td>
<td>r3</td>
</tr>
<tr>
<td>7</td>
<td>INDIRD</td>
<td>6</td>
<td>2</td>
<td></td>
<td></td>
<td>r2</td>
<td>r3</td>
</tr>
<tr>
<td>8</td>
<td>ADDRGP</td>
<td>.b</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>r5</td>
</tr>
<tr>
<td>9</td>
<td>ADDP</td>
<td>4 8</td>
<td>1</td>
<td></td>
<td></td>
<td>r2</td>
<td>r5</td>
</tr>
<tr>
<td>10</td>
<td>INDIRD</td>
<td>9</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>r2</td>
</tr>
<tr>
<td>11</td>
<td>ADDD</td>
<td>7 10</td>
<td>1</td>
<td></td>
<td></td>
<td>r3</td>
<td>r4</td>
</tr>
<tr>
<td>12</td>
<td>SUBD</td>
<td>7 10</td>
<td>1</td>
<td></td>
<td></td>
<td>r3</td>
<td>r4</td>
</tr>
<tr>
<td>13</td>
<td>MULD</td>
<td>11 12</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>CVDI</td>
<td>13</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>ASGNI</td>
<td>1 14</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Nodes with count fields greater than 1 represent four common subexpressions: the lvalue of i (node 1), the addressing expression i<<3 (node 4), and the rvalues of a[i] (node 7) and b[i] (node 10). If only registers 1–5 are available, ralloc runs out of registers at node 10. The linearized forest at that point is shown to the right of the vertical line in the display above. As indicated by the non-zero counts and the sets column, node 1 is using r1 and node 7 is using r3 and r4. Node 10 needs a register pair, but only registers r2 and r5 are available. Note that the count of node 4, which is i<<3, has dropped to 0 because ralloc has processed both uses (nodes 6 and 9).

getreg calls spillee to identify a register pair to be spilled so that it can be used for node 10. spillee chooses the register whose next use is the most distant in the linearized forest. This choice is analogous to the optimal demand paging strategy that replaces pages whose next use is most distant in the execution stream [7].

For each register, spillee simply searches down the linearized forest for register uses and records the most distant.

static int spillee(Node dot, unsigned m) {
    int bestdist = -1, bestreg = 0, dist, r;
Node q;

for (r = 1; r < nregs - (m>>1); r++) {
    dist = 0;
    for (q = dot->x.next; q && !(uses(q)&(m<<r)); q = q->x.next)
        dist++;
    if (dist > bestdist) {
        bestdist = dist;
        bestreg = r;
    }
}
return bestreg;

spillee is called with dot equal to the node at which the spill is required, which is node 10 in the example above. uses(p) returns a bit mask giving the registers used by node p, i.e., the registers set by p’s kids:

static unsigned uses(Node p) {
    int i;
    unsigned m = 0;

    for (i = 0; i < MAXKIDS; i++)
        if (p->kids[i])
            m |= sets(p->kids[i]);
    return m;
}

In this example, spillee finds r1 used in node 15, which is at ‘distance’ 4, and r3 and r4 used in node 11 at distance 0. Consequently, spillee returns r1, which denotes both r1 and r2 because m is 3.

The implementations of spillee above and spill below assume that the instruction emitted for each node reads the registers it uses before setting the register allocated to it. This assumption permits ralloc to reassign registers used by a node to that node. It also permits spillee to start its scan after the current instruction. This assumption is invalid on machines with two-address instructions.

Actually spilling the register chosen by spillee and inserting the reloads is done by spill and its associates, genspill and genreloads. In the example, these functions collaborate to ‘break the dag’ at node 1, which sets register r1 (r2 is already free). genspill generates a temporary and the nodes necessary to store the value computed by node 1 into the temporary. These new nodes are stitched into the linearized forest immediately after node 1. genreloads changes future uses of the value computed by node 1 to reload the value from the temporary instead of referencing node 1 directly. The effect is to remove the common subexpression by assigning it to a temporary.
spill identifies the locations at which to insert the spills by searching the linearized forest beyond dot for nodes that use the registers that are to be spilled, e.g., r1 in the example.

```c
static void spill(int r, unsigned m, Node dot) {
    int i;
    Node p = dot;

    while (p = p->x.next)
        for (i = 0; i < MAXKIDS; i++)
            if (p->kids[i] & & sets(p->kids[i]) & (m<<r)) {
                Symbol temp = genspill(p->kids[i]);
                rmask & = ~sets(p->kids[i]);
                genreloads(dot, p->kids[i], temp);
            }
}
```

genspill allocates the temporary, generates the spill code, and inserts it into the linearized forest right after the node that set the spilled. It returns the symbol table entry for the temporary so that it can be used by genreloads to generate the reloads.

```c
static Symbol genspill(Node p) {
    Symbol temp = newtemp(AUTO, typecode(p));
    Node q = p->x.next;

    linearize(newnode(ASGN + typecode(p),
        newnode(ADDRLP, 0, 0, temp), p, 0),
        &p->x.next, p->x.next);
    rmask & = ~1;
    for (p = p->x.next; p != q; p = p->x.next)
        ralloc(p);
    rmask |= 1;
    return temp;
}
```
typecode is like optype, but maps U to I because the front-end uses ASGN and INDIRI to store and load unsigned values. It also maps B to P because registers always point structures. The front-end's newtemp(class, type) allocates a new temporary or reuses an existing one if its storage class and type code match those requested, and it announces a new temporary by calling local as usual.

The spill code is an ADDRLP node to compute the address of the temporary and an ASGN node to store the value into that address. The right operand of the ASGN is simply p, the node that set the registers to spill, e.g., node l above. A
node is allocated and initialized by the front-end function `newnode(op, l, r, sym)`, where `op` is the operator, `l` and `r` are `kids[0]` and `kids[1]`, and `sym` is the symbol table pointer for leaf nodes. `newnode` also increments the reference counts of `l` and `r`.

Finally, `genspil` linearizes the spill code and inserts it right after `p`, which sets the spilled register. The manipulation of `rmask` ensures that `ralloc` assigns register `r0` — which is not otherwise allocated — to the `ADDRLP` node to compute the address. The linearized forest after `genspil` returns is

```
<table>
<thead>
<tr>
<th>#</th>
<th>op</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ADDRGP</td>
<td>_i</td>
<td>1</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>ADDRLP</td>
<td>-4(fp)</td>
<td>0</td>
<td>r0</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>ASGNP</td>
<td>16</td>
<td>1</td>
<td>0</td>
<td>r0</td>
</tr>
<tr>
<td></td>
<td>IINDIR</td>
<td>1</td>
<td>0</td>
<td>r1</td>
<td>r2</td>
</tr>
</tbody>
</table>
```

Nodes 16 and 17 are the spill code. This code references node 1, so its `count` goes to 2 momentarily until `ralloc` processes the reference from the spill code. The remaining reference is from node 15. Node 16's symbol `-4(fp)` is the temporary location.

After `genspil` returns the temporary, `spill` frees the registers set by the node and calls `genreloads`. `genreloads` searches the linearized forest after `dot` for uses of `p`.

```c
static void genreloads(Node dot, Node p, Symbol temp) {
    int i;
    Node last;

    for (last = dot; dot = dot->x.next; last = dot)
        for (i = 0; i < MAXKIDS; i++)
            if (dot->kids[i] == p) {
                dot->kids[i] = newnode(INDIR + typecode(p),
                                       newnode(ADDRLP+P, 0, 0, temp), 0, 0);
                dot->kids[i]->count = 1;
                p->count--;
                linearize(dot->kids[i], &last->x.next, last->x.next);
                last = dot->kids[i];
            }
}
```

Each use of `p` is changed to a reload of the temporary `temp`. The reload code is an `ADDRLP` node to compute the address of the temporary and an `INDIR` node to load the value from that address. There is only one reference to each reload, so the `INDIR`'s `count` is set accordingly, and `p->count` is decremented to reflect the change. The reload code is linearized and inserted into the linearized node
list immediately before its use. For the example above, a reload is placed before node 15. Since the reload is beyond the point that gen has reached, registers are allocated for these nodes by subsequent calls to ralloc.

The linearized forest after spill returns is

<table>
<thead>
<tr>
<th>#</th>
<th>op</th>
<th>kids</th>
<th>sym</th>
<th>count</th>
<th>uses</th>
<th>sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADDRGP</td>
<td>_i</td>
<td></td>
<td>0</td>
<td></td>
<td>r1</td>
</tr>
<tr>
<td>16</td>
<td>ADDRLP</td>
<td>-4(fp)</td>
<td>0</td>
<td></td>
<td></td>
<td>r0</td>
</tr>
<tr>
<td>17</td>
<td>ASGWP</td>
<td>16 1</td>
<td></td>
<td>0</td>
<td>r0 r1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>INDIRI</td>
<td>1</td>
<td></td>
<td>0</td>
<td>r1 r2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>CVDI</td>
<td>13 1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>ADDRLP</td>
<td>-4(fp)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>INDIRP</td>
<td>18 1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>ASGNI</td>
<td>19 14</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Nodes 18 and 19 are the reload. Note that node 1’s count has become 0; after inserting the reload, there are no unprocessed references.

Another spill occurs at node 11, which computes a[i]+b[i]. Registers r1 and r2, which are set by node 10, are spilled, and node 12 is edited to refer to the second temporary, which is reloaded by nodes 22 and 23 inserted before node 12. The last spill occurs at node 23, which is the reload created for the second spill. r1 and r2, which are set by node 11, are spilled again, and node 13 is edited to refer to the third temporary. The final linearized forest and corresponding VAX code follows; count fields are omitted because they’re all 0, and each instruction shows which registers are used and set by each node.

VAX instruction      | #  | op          | kids | sym |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>moval _i,r1</td>
<td>1</td>
<td>ADDRGP</td>
<td>_i</td>
<td></td>
</tr>
<tr>
<td>moval -4(fp),r0</td>
<td>16</td>
<td>ADDRLP</td>
<td>-4(fp)</td>
<td></td>
</tr>
<tr>
<td>movl r1,(r0)</td>
<td>17</td>
<td>ASGWP</td>
<td>16 1</td>
<td></td>
</tr>
<tr>
<td>movl (r1),r2</td>
<td>2</td>
<td>INDIRI</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>movl $3,r3</td>
<td>3</td>
<td>CMTSTI</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ashl r3,r2,r2</td>
<td>4</td>
<td>LSHI</td>
<td>2 3</td>
<td></td>
</tr>
<tr>
<td>moval _a,r3</td>
<td>5</td>
<td>ADDRGP</td>
<td>_a</td>
<td></td>
</tr>
<tr>
<td>addl3 r3,r2,r3</td>
<td>6</td>
<td>ADDP</td>
<td>4 5</td>
<td></td>
</tr>
<tr>
<td>movd (r3),r3</td>
<td>7</td>
<td>INDIRD</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>moval _b,r5</td>
<td>8</td>
<td>ADDRGP</td>
<td>_b</td>
<td></td>
</tr>
<tr>
<td>addl3 r5,r2,r2</td>
<td>9</td>
<td>ADDP</td>
<td>4 8</td>
<td></td>
</tr>
<tr>
<td>movd (r2),r1</td>
<td>10</td>
<td>INDIRD</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>moval -12(fp),r0</td>
<td>20</td>
<td>ADDRLP</td>
<td>-12(fp)</td>
<td></td>
</tr>
<tr>
<td>movd r1,(r0)</td>
<td>21</td>
<td>ASGND</td>
<td>20 10</td>
<td></td>
</tr>
<tr>
<td>addd3 r1,r3,r1</td>
<td>11</td>
<td>ADDDD</td>
<td>7 10</td>
<td></td>
</tr>
<tr>
<td>moval -20(fp),r0</td>
<td>24</td>
<td>ADDRLP</td>
<td>-20(fp)</td>
<td></td>
</tr>
<tr>
<td>movd r1,(r0)</td>
<td>25</td>
<td>ASGND</td>
<td>24 11</td>
<td></td>
</tr>
</tbody>
</table>
moval -12(fp),r5 22. ADDRLP  -12(fp)
movd (r6),r1 23. INDIRD 22
subd3 r1,r3,r1 12. SUBD  7 23
moval -20(fp),r3 26. ADDRLP  -20(fp)
movd (r3),r3 27. INDIRD 26
muld3 r1,r3,r1 13. MULD  27 12
cvt dl r1,r1 14. CVDI  13
moval -4(fp),r2 18. ADDRLP  -4(fp)
movl (r2),r2 19. INDIRP 18
movl r1,(r2) 15. ASGNI  19 14

Spills have added nodes 16–27.

7 Discussion

lcc’s code generation interface is smaller than most because it omits the inessential and makes simplifying assumptions. These omissions and assumptions do, however, limit the interface’s applicability to other languages and machines.

The datatype assumptions detailed in Section 2, e.g., that un signed, integers, and long integers all have the same size, make it possible to have only 9 type suffixes and 111 type-specific operators. Relaxing these assumptions would increase this vocabulary; e.g., adding a suffix for long doubles would also add at least 19 more operators.

The interface assumes that all pointer types have the same representation, which precludes its use for word-addressed machines. Differentiating between character and word pointers would add another suffix and at least 13 more operators.

The operator repertoire omits some operators whose effect can be synthesized from simpler ones. For example, bit fields are accessed with shifting and masking instead of specific bit-field operators, so code quality may suffer on machines with bit-field instructions. The front end special-cases one-bit fields and generates efficient masking dags, which often yields better code than code that uses bit-field instructions.

The front end implements switch statements completely. It generates a binary search of dense branch tables; inline comparisons replace small tables [2]. It fabricates the tables and indirect jumps using the functions global and defaddress and the JUMPV operator. This decision prevents back ends from using larger ‘case’ instructions, which usually combine a bounds check and an indirect branch through a table, but these instructions are increasingly rare, and some don’t suit C anyway. For example, the branch table for the VAX’s casel instruction is limited to 16-bit offsets.

The interface has no direct support for building a flow graph and other structures that facilitate global optimization. More elaborate versions of function
and gen could collaborate to build the relevant structures, perform optimizations, and invoke the simpler gen. The front end's gencode and emitcode traverse an approximation of a flow graph; future work may focus on a machine-independent optimizer that edits this graph while preserving the current interface.

To date, the interface has been used only for ANSI C. But it has little that is specific to C, and it could be used for similar languages and perhaps for languages with features like nested procedures, objects, and exception handling. Existing compilers for some object-oriented languages with these features, such as C++, Modula-3, and Eiffel, generate C, so, in principle, the interface could be used for these languages.

References


A Interface Functions

In the table below, † flags those interface functions that are implemented by macros in the sample.

<table>
<thead>
<tr>
<th>Section</th>
<th>Interface Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>void address(Symbol, Symbol, int)</td>
</tr>
<tr>
<td>5.3</td>
<td>void asmcode(char *, Symbol [])</td>
</tr>
<tr>
<td>4.4</td>
<td>void blockbeg(Env *, e)</td>
</tr>
<tr>
<td>4.4</td>
<td>void blockend(Env *, e)</td>
</tr>
<tr>
<td>† 3.6</td>
<td>void defaddress(Symbol)</td>
</tr>
<tr>
<td>3.5</td>
<td>void defconst(int, Value)</td>
</tr>
<tr>
<td>3.6</td>
<td>void defstring(int, char *)</td>
</tr>
<tr>
<td>3.1</td>
<td>void defsymbol(Symbol)</td>
</tr>
<tr>
<td>5.2</td>
<td>void emit(Node)</td>
</tr>
<tr>
<td>† 3.2</td>
<td>void export(Symbol)</td>
</tr>
<tr>
<td>4.1</td>
<td>void function(Symbol, Symbol [], Symbol [], int)</td>
</tr>
<tr>
<td>5.1</td>
<td>Node gen(Node)</td>
</tr>
<tr>
<td>3.4</td>
<td>void global(Symbol)</td>
</tr>
<tr>
<td>† 3.2</td>
<td>void import(Symbol)</td>
</tr>
<tr>
<td>4.2</td>
<td>void local(Symbol)</td>
</tr>
<tr>
<td>2.1</td>
<td>void progbeg(int, char *[])</td>
</tr>
<tr>
<td>† 2.1</td>
<td>void progend(void)</td>
</tr>
<tr>
<td>3.3</td>
<td>void segment(int)</td>
</tr>
<tr>
<td>† 3.6</td>
<td>void space(int)</td>
</tr>
</tbody>
</table>

B Front-End Functions

void *alloc(int n) permanently allocates n bytes and returns a pointer to the first byte.

void fatal(char *name, char *fmt, ...) announces a compiler error in module name on standard error, prints up to 4 arguments, and terminates execution. See printf for formatting details.

void fprintf(int fd, char *fmt, ...) prints up to 5 arguments on the file descriptor fd. See printf for formatting details. If fd is not 1 (standard output), fprintf flushes the output buffer for fd via fflush.

Type freturn(Type t) is the type of the return value for function type t.

int generic(int op) is the generic version of the type-specific operator op.

int genlabel(int n) increments the generated identifier counter by n and returns its previous value.
int istype(Type t) are a set of type predicates that return non-zero if type t is the type in the following table.

<table>
<thead>
<tr>
<th>predicate</th>
<th>type</th>
<th>predicate</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>isarith</td>
<td>arithmetic</td>
<td>isint</td>
<td>integral</td>
</tr>
<tr>
<td>isarray</td>
<td>array</td>
<td>isptr</td>
<td>pointer</td>
</tr>
<tr>
<td>ischar</td>
<td>character</td>
<td>iscalar</td>
<td>scalar</td>
</tr>
<tr>
<td>isdouble</td>
<td>double</td>
<td>istruct</td>
<td>structure or union</td>
</tr>
<tr>
<td>isenum</td>
<td>enumeration</td>
<td>isunion</td>
<td>union</td>
</tr>
<tr>
<td>isfloat</td>
<td>floating</td>
<td>isunsigned</td>
<td>unsigned</td>
</tr>
<tr>
<td>isfunc</td>
<td>function</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Node newnode(int op, Node l, Node r, Symbol sym) allocates a node and initializes the op field to op, kids[0] and kids[1] to l and r, and sym[0] to sym and returns a pointer to the new node. If l and r are non-null, their count fields are incremented.

Symbol newconst(Value v, int t) installs an constant with value v and type suffix t into the symbol table, if necessary, and returns a pointer to the symbol table entry.

Symbol newtemp(int class, int t) creates a temporary with storage class class and a type synonymous with type suffix t and returns a pointer the symbol table entry. If an existing temporary with the appropriate class and type is available, it is used; otherwise, the new temporary is announced by calling local.

void outflush(void) writes the current output buffer to the standard output, if it's not empty.

void outs(char *s) appends string s to the output buffer for standard output and calls outflush if the resulting buffer pointer is within 80 characters of the end of the buffer.

int optype(op) is the type suffix of the type-specific operator op.

void print(char *fmt, ...) prints up to 5 arguments on standard output similar to printf. The supported formats are %c, %d, %o, %x, and %s, and precision and field width specifications are not supported. print calls outflush if it prints a newline character from fmt within 80 characters of the end of the output buffer, and each format except %c does the actual output with outs, which may also flush the buffer.

int roundup(int n, int m) is n rounded up to the next multiple of m where m is a power of 2.
void sprintf(char *s, char *fmt, ...) formats up to 5 arguments into string s. See print for formatting details.

char *string(char *s) installs s in the string space, if necessary, and returns a pointer to the installed copy.

char *stringd(int n) returns the string representation of n; the returned string is installed in the string space by string.

char *stringf(char *fmt, ...) formats up to 5 arguments into a string in the string space and returns a pointer to that string. See print for formatting details.

void *talloc(int n) temporarily allocates n bytes and returns a pointer to the first byte. The storage is released at the end of the current function.

int ttop(Type t) is the type suffix synonymous with type t.

int variadic(Type t) is true if type t denotes a variadic function.