The lcc 4.x Code-Generation Interface

Christopher W. Fraser and David R. Hanson
Microsoft Research
cwfraser@microsoft.com  drh@microsoft.com

July 2001

Technical Report
MSR-TR-2001-64

Abstract

Lcc is a widely used compiler for Standard C described in A Retargetable C Compiler: Design and Implementation. This report details the lcc 4.x code-generation interface, which defines the interaction between the target-independent front end and the target-dependent back ends. This interface differs from the interface described in Chap. 5 of A Retargetable C Compiler. Additional information about lcc is available at http://www.cs.princeton.edu/software/lcc/.
The lcc 4.x Code-Generation Interface

1. Introduction

Lcc is a widely used compiler for Standard C described in A Retargetable C Compiler [1]. Version 4.x is the current release of lcc, and it uses a different code-generation interface than the interface described in Chap. 5 of Reference 1. This report details the 4.x interface. Lcc distributions are available at http://www.cs.princeton.edu/software/lcc/ along with installation instructions [2].

The code generation interface specifies the interaction between lcc’s target-independent front end and target-dependent back ends. The interface consists of a few shared data structures, a 33-operator language, which encodes the executable code from a source program in directed acyclic graphs, or dags, and 18 functions, that manipulate or modify dags and other shared data structures. On most targets, implementations of many of these functions are very simple. The front and back ends share some fields of the shared data structures, but other fields are private to the front or back end. In the descriptions below, fields that are private to the front end or are not explained here are dimmed.

The front and back ends are packaged together in a single, monolithic program, and they are clients of each other. The front end calls 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. The front-end functions that back ends may call are summarized in the sections below.

2. Interface Records

A cross-compiler produces code for one machine while running on another. Lcc can be linked with code generators for several targets, so it can be used as either a native compiler or a cross-compiler. lcc’s interface record captures everything that its front end needs to know about a target machine, including pointers to the interface routines, type metrics, and interface flags. The interface record is defined in Figure 1.

The interface record has fields for type metrics, interface flags, and pointers to interface functions. The x field is an extension in which the back end stores target-specific interface data and functions. The x field is private to the back end and is defined in config.h.

lcc has a distinct instance of the interface record for each target, but interface records can share function pointers.

A type metric specifies the size and alignment for a primitive type:

```
typedef struct metrics {
    unsigned char size, align, outofline;
} Metrics;
```

The outofline flag controls the placements of constants of the associated type. If outofline is one, constants cannot appear in dags; such constants are placed in anonymous static variables and their values are accessed by fetching the variables. Each primitive type has a metric field in the interface record, as shown in Figure 1. ptrmetric describes pointers of all types. The alignment of a structure is the maximum of the alignments of its fields and structmetric.align, which thus gives the minimum alignment for structures; structmetric.size is unused. Back ends usually set outofline to zero only for those types whose values can appear as immediate operands of instructions.
typedef struct interface {
    Metrics charmetric;
    Metrics shortmetric;
    Metrics intmetric;
    Metrics longmetric;
    Metrics longlongmetric;
    Metrics floatmetric;
    Metrics doublemetric;
    Metrics longdoublemetric;
    Metrics ptrmetric;
    Metrics structmetric;
    unsigned little_endian:1;
    unsigned mulops_calls:1;
    unsigned wants_callb:1;
    unsigned wants_argb:1;
    unsigned left_to_right:1;
    unsigned wants_dag:1;
    unsigned unsigned_char:1;
    void (*address)(Symbol, Symbol, long);
    void (*blockbeg)(Env *);
    void (*blockend)(Env *);
    void (*defaddress)(Symbol);
    void (*defconst)(int, int, Value);
    void (*defstring)(int n, char *);
    void (*defsymbol)(Symbol);
    void (*emit)(Node);
    void (*export)(Symbol);
    void (*function)(Symbol, Symbol[], Symbol[], int);
    Node (*gen)(Node);
    void (*global)(Symbol);
    void (*import)(Symbol);
    void (*local)(Symbol);
    void (*progbeg)(int, char *[]);
    void (*progend)(void);
    void (*segment)(int);
    void (*space)(int);
    void (*stabblock)(int, int, Symbol*);
    void (*stabend)(Coordinate *, Symbol, Coordinate **, Symbol *, Symbol *);
    void (*stabfend)(Symbol, int);
    void (*stabinit)(char *, int, char *[]);
    void (*stabline)(Coordinate *);
    void (*stabsym)(Symbol);
    void (*stabtype)(Symbol);
    Xinterface x;
} Interface;

Figure 1. The interface record.

The size and alignment for characters must be one. The front end correctly treats signed and unsigned integers and longs as distinct types, but it assumes that they all share intmetric. Likewise for doubles and long doubles.

The 18 interface functions are described in sections 7–10. These function pointers are often denoted by just their name. For example, gen is used instead of the more accurate but verbose ‘the function pointed to by the gen field of the interface record.’

The interface record also holds pointers to some functions that the front end calls to emit symbol tables for debuggers. This report does not describe these ‘stab’ functions.
3. Symbols

A symbol represents a variable, label, type, or constant and is defined in Figure 2. The scope and

typedef struct symbol *Symbol;

struct symbol {
    char *name;
    int scope;
    Coordinate src;
    Symbol up;
    List uses;
    int sclass;
    unsigned structarg:1;
    unsigned addressed:1;
    unsigned temporary:1;
    unsigned computed:1;
    unsigned generated:1;
    unsigned defined:1;
    Type type;
    float ref;
    union {
        ...
        See Figure 3...
    } u;
    Xsymbol x;
}

Figure 2. The symbol structure.

sclass fields identify the kind of symbol. For variables and constants, the back end may query
the type field to learn the data type suffix of the item. For variables and labels, the floating-
point value of the ref field approximates the number of times that variable or label is referred-
enced; a nonzero value thus indicates that the variable or label is referenced at least once. For
labels, constants, and some variables, a field of the union u supplies additional data as detailed
below.

Variables have a scope equal to GLOBAL, PARAM, or LOCAL k for nesting level k. sclass 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. For global and
static variables, u.seg gives the logical segment in which the variable is defined. If the interface
flag wants_dag is zero, the front end generates explicit temporary variables to hold common
subexpressions—those used more than once. It sets the u.t.cse fields of these symbols to the
dag nodes that compute the values stored in them.

The flags temporary and generated are set for temporaries, and the flag generated is set
for labels and other generated variables, like those that hold string literals. computed is set if the
symbol was defined by calling the interface function address. structarg identifies structure
parameters when the interface flag wants_argb is set; the material below on wants_argb elaborates.
defined is set when the front end announces a symbol to the back end via the interface
functions defsymbol, address, local, or function.

Labels have a scope equal to LABELS. The u.l.label field is a unique numeric value that
identifies the label, and name is the string representation of that value. Labels have no type or
sclass.

Constants have a scope equal to CONSTANTS, and an sclass equal to STATIC. For an integral
or pointer constant, name is its string representation as a C constant. For other types, name is
undefined. The actual value of the constant is stored in the u.c.v field, which is defined by:
typedef union value {
    long i;
    unsigned long u;
    long double d;
    void *p;
    void (*g)(void);
} Value;

The type of a constant indicates which Value field holds the actual value. If a variable is generated to hold the constant, u.c.loc points to the symbol-table entry for that variable.

The src field gives the source location at which the symbol is defined as a 'source coordinate:'

typedef struct coord {
    char *file;
    unsigned x, y;
} Coordinate;
	src.y is the line number and src.x is the character number within that line; both are zero-based.
Symbols have an x field with type Xsymbol, defined in config.h. It’s an extension in which the back end stores target-specific data for the symbol, like the stack offset for locals. The x field is private to the back end, and thus its contents are not part of the interface.

3.1 Symbol Constructors

Symbol constant(Type ty, Value v)
installs a constant with type ty and value v into the symbol table with a scope of CONSTANTS, if necessary, and returns a pointer to the symbol-table entry. ty must be a INT, UNSIGNED, FLOAT, ARRAY, POINTER, or FUNCTION type with the corresponding value in v.i, v.u, v.d, v.p, v.p, or v.g.

Symbol findlabel(int label)
returns a symbol-table entry for the label label, creating it, if necessary. The entry has scope LABELS.

Symbol intconst(int n)
calls constant to install the int constant n into the symbol table.

Symbol newtemp(int sclass, int suffix, int size)
creates a temporary with storage class sclass and the type returned by btot(suffix), and returns a pointer to the symbol-table entry. The new temporary is announced by calling local.

4. Types

Symbols have a type field. If the symbol represents a constant or variable, the type field points to a structure that describes the type of the item:

```c
typedef struct type *Type;

struct type {
    int op;
    Type type;
    int align;
    int size;
    union {
        Symbol sym;
        struct {
            unsigned oldstyle:1;
            Type *proto;
        } f;
    } u;
    Xtype x;
};
```

The op field holds an integer operator code, and the type field holds the operand. The operators are the values of the following global enumeration constants or combinations of their values:

```
INT UNSIGNED FLOAT ENUM
ARRAY STRUCT UNION
POINTER FUNCTION
VOID
CONST
```
The align and size fields give the type's alignment and the size of objects of that type in bytes. The size must be a multiple of the alignment. The back end must allocate space for a variable so that its address is a multiple of its type's alignment. The size field can be zero, which is an incomplete type.

The INT, UNSIGNED, and ENUM operators define the integral types, and the FLOAT operator defines the floating types. Together, these types are known as the arithmetic types. The various C arithmetic types are represented by these types with the sizes and alignments taken from the type metrics in the interface record. For example, the C type 'char' is represented by a type with an INT op field and size and align fields equal to one. The front end initializes and exports types for the C built-in types:

```c
typedef struct field *Field;
struct field {
    char *name;
};
```

Except for ENUM types, the u.sym field of an arithmetic type points to a GLOBAL symbol for the type. u.sym->name is the C type name, u.sym->type is the type structure, u.sym->addressed is the outofline field of the relevant metric, and u.sym->u.limits.min and u.sym->u.limits.max give the minimum and maximum values of the type.

Except for ENUM types, the arithmetic types have no operands. The type field of an ENUM type is its compatible integral type, i.e., the type of the enumeration identifiers. For lcc, this type is always int. The u.sym field points to a symbol for its tag. u.sym->name is the tag, u.sym->type is the enumeration type, and u.sym->u.idlist points to a null-terminated array of symbols for the enumeration constants associated with the enumeration type. For each u.sym->u.idlist[i], sclass is ENUM, type is the enumeration type, and u.value is the integral value of the enumeration constant.

The ARRAY, STRUCT, and UNION operators identify the aggregate types. ARRAY's operand is the element type.

STRUCT and UNION do not have operands; u.sym points to a symbol that describes the fields. u.sym->name is tag and u.sym->u.s->fields is a null-terminated linked list of field structures, one for each field in the order the fields appear in the structure:
Type type;
int offset;
short bitsize;
short lsb;
Field link;
}

name holds the field name, type is the field's type, and offset is the byte offset to the field in an instance of the structure. When a field describes a bit field, the type field is either inttype or unsignedtype, because those are the only two types allowed for bit fields. The lsb field is nonzero and is the number of the least significant bit in the bit field plus one, where bit numbers start at zero with the least significant bit. This bit field representation for bit fields does not depend on the target's endianness; the same representation is used for both big and little endians. The flags u.sym->u.s.cfields and u.sym->vfields are one if the structure or union type has any const-qualified or volatile-qualified fields.

The POINTER operator defines pointer types, and the operand gives the referenced type. The front end initializes and exports the following types:

Type charptype
Type funcptype
Type voidptype
Type unsignedptr
Type signedptr

funcptype and voidptype are the C types "void (*)(void)" and "void*," which are the putative function pointer and object pointer. unsignedptr and signedptr are UNSIGNED and INT types whose sizes and alignments are taken from IR->ptrrmetric.

The FUNCTION operator defines function types. The operand is the return type, and u.f.old-style is one if the type definition is "old-style," that is, without a prototype. If the function type is defined with a prototype, u.f.proto points to a null-terminated array of types for each formal parameter. If the function type has a variable number of arguments, the array consists of the types of the declared arguments, voidtype, and the terminating null.

The VOID operator identifies the void type; it has no operand and its size and alignment are both zero. The front end initializes and exports the single instance of the void type:

Type voidtype

The CONST and VOLATILE operators specify qualified types; their operands are the unqualified versions of the types. The sum CONST+VOLATILE is also a type operator, and it specifies a type that is both constant and volatile.

The x field plays the same role for types as it does in symbols; back ends may define Xtype to add target-specific fields to the type structure. This facility is most often used to support debuggers.

4.1 Type Constructors

Type array(Type ty, int n, int align)
    returns a type for an 'array of n elements of type ty.' If align is nonzero, it is used for the alignment instead of ty->align.

Type atop(Type ty)
    returns a type for a pointer to element type of array type ty.
Type func(Type ty, Type *proto, int oldstyle)
  returns a type for a function that returns type ty and has the specified prototype. If oldstyle
  is one, proto may be null.

Type ptr(Type ty)
  returns a type for 'pointer to ty.'

Type qual(int op, Type ty)
  returns a qualified version of type ty. op must be CONST or VOLATILE.

4.2 Operations on Types
The innards of types are revealed so that back ends may read the size and align fields, and
read and write the x fields, and pass the type field to the functions defined in this section. By
convention, these are the only fields the back ends are allowed to inspect. In practice, however,
back ends may also need to inspect the u fields to, for example, examine structure fields and
function prototypes.

int hasproto(Type ty)
  returns 1 of the function type ty has a prototype and 0 otherwise.

Type btot(int suffix, int size)
  returns a type of size bytes that corresponds to the node type suffix. See ttob and Sec. 5.

Type deref(Type ty)
  returns the reference type of pointer type ty.

Type freturn(Type ty)
  return the return type of the function type ty.

int isvolatile(Type t)
int isconst(Type ty)
int isarray(Type ty)
int isstruct(Type ty)
int isunion(Type ty)
int isfunc(Type ty)
int isptr(Type ty)
int ischar(Type ty)
int isint(Type ty)
int isfloat(Type ty)
int isarith(Type ty)
int isunsigned(Type ty)
int isscalar(Type ty)
int isenum(Type ty)
  Each of these type predicates returns 1 if ty is the type indicated by its name and 0 otherwise.

int ttob(Type ty)
  returns the node operator type suffix that corresponds to type ty. See btot and Sec. 5.

Type unqual(Type ty)
  returns the unqualified version of type ty.

int variadic(Type ty)
  returns 1 if function type ty has a variable number of arguments.

5. Dag Operators
Executable code is specified by dags. A function body is a sequence, or forest, of dags, each of
which is passed to the back end via gen. Dag nodes, sometimes called nodes, are defined by:

```c
typedef struct node *Node;

struct node {
    short op;
    short count;
    Symbol syms[3];
    Node kids[2];
    Node link;
    Xnode x;
};
```

The elements of kids point to the operand nodes. Some dag operators also take one or two symbol-table pointers as operands; these appear in syms. The back end may use the third syms for its own purposes; the front end uses it, too, but its uses are temporary and occur before dags are passed to the back end. The link field points to the root of the next dag in the forest.

The count field records the number of times the value of this node is used or referred to by others. Only references from kids count; link references don't count because they don't represent a use of the value of the node. Indeed, link is meaningful only for root nodes, which are executed for side effect, not value. If the interface flag wants_dag is zero, roots always have a zero count. The generated code for shared nodes—those whose count exceeds one—must evaluate the node only once; the value is used count times.

The x field is the back end's extension to nodes. The back end defines the type Xnode in config.h to hold the per-node data that it needs to generate code.

The op field a of node structure holds a dag operator, which consists of a generic operator, a type suffix, and a size indicator. The type suffixes are:

```c
enum {
    F=FLOAT,
    I=INT,
    U=UNSIGNED,
    P=POINTER,
    V=VOID,
    B=STRUCT
};
```

Given a generic operator o, a type suffix t, and a size modifier s, a type- and size-specific operator is formed by o+t+sizeop(s). For example, ADD+F+sizeop(4) forms the operator ADDF4, which denotes the sum of two 4-byte floats. Similarly, ADD+F+sizeop(8) forms ADDF8, which denotes 8-byte floating addition.

Table 1 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 variants. The notations in the syms column give the number of syms values and a one-letter code that suggests their uses: 1V indicates that syms[0] points to a symbol for a variable, 1C indicates that syms[0] is a constant, and 1L indicates that syms[0] is a label. For 1S, syms[0] is a constant whose value is a size in bytes; 2S adds syms[1], which is a constant whose value is an alignment. Finally, 1T indicates the syms[0]->type is the only field of interest.

The entries in the Sizes column indicate sizes of the operators that back ends must implement. Letters denote the size of float (f), double (d), long double (x), character (c), short integer (s), integer (i), long integer (l), 'long long' integer (h), and pointer (p). These sizes are separated
**Table 1.** Node operators.

<table>
<thead>
<tr>
<th>syms</th>
<th>kids</th>
<th>Operator</th>
<th>Type Suffixes</th>
<th>Sizes</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1V</td>
<td>0</td>
<td>ADDRF</td>
<td>...P...</td>
<td>p</td>
<td>address of a parameter</td>
</tr>
<tr>
<td>1V</td>
<td>0</td>
<td>ADDRGl</td>
<td>...P...</td>
<td>p</td>
<td>address of a global</td>
</tr>
<tr>
<td>1V</td>
<td>0</td>
<td>ADDRL</td>
<td>...P...</td>
<td>p</td>
<td>address of a local</td>
</tr>
<tr>
<td>1C</td>
<td>0</td>
<td>CNST</td>
<td>FIUP...</td>
<td>fdx</td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>BCOM</td>
<td>.IU... ilh</td>
<td></td>
<td>bitwise complement</td>
</tr>
<tr>
<td>1S</td>
<td>1</td>
<td>CVF</td>
<td>FI...</td>
<td>fdx ilh</td>
<td>convert from float</td>
</tr>
<tr>
<td>1S</td>
<td>1</td>
<td>CVI</td>
<td>FIU...</td>
<td>fdx csilh csilhp</td>
<td>convert from signed integer</td>
</tr>
<tr>
<td>1S</td>
<td>1</td>
<td>CVP</td>
<td>..U... p</td>
<td></td>
<td>convert from pointer</td>
</tr>
<tr>
<td>1S</td>
<td>1</td>
<td>CVU</td>
<td>.IUP... csilh p</td>
<td></td>
<td>convert from unsigned integer</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>INDIR</td>
<td>FIUP.B</td>
<td>fdx</td>
<td>fetch</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>NEG</td>
<td>FI...</td>
<td>fdx ilh</td>
<td>negation</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>ADD</td>
<td>FIU...</td>
<td>fdx ilh ilhp p</td>
<td>addition</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>BAND</td>
<td>.IU... ilh</td>
<td></td>
<td>bitwise AND</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>BOR</td>
<td>.IU... ilh</td>
<td></td>
<td>bitwise inclusive OR</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>BXOR</td>
<td>.IU... ilh</td>
<td></td>
<td>bitwise exclusive OR</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>DIV</td>
<td>FIU...</td>
<td>fdx ilh</td>
<td>division</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>LSH</td>
<td>.IU... ilh</td>
<td></td>
<td>left shift</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>MOD</td>
<td>.IU... ilh</td>
<td></td>
<td>modulus</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>MUL</td>
<td>FIU...</td>
<td>fdx ilh</td>
<td>multiplication</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>RSH</td>
<td>.IU... ilh</td>
<td></td>
<td>right shift</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>SUB</td>
<td>FIU...</td>
<td>fdx ilh ilhp p</td>
<td>subtraction</td>
</tr>
<tr>
<td>2S</td>
<td>2</td>
<td>ASGN</td>
<td>FIUP.B</td>
<td>fdx</td>
<td>assignment</td>
</tr>
<tr>
<td>1L</td>
<td>2</td>
<td>EQ</td>
<td>FIU...</td>
<td>fdx ilh ilhp</td>
<td>jump if equal</td>
</tr>
<tr>
<td>1L</td>
<td>2</td>
<td>GE</td>
<td>FIU...</td>
<td>fdx ilh ilhp</td>
<td>jump if greater than or equal</td>
</tr>
<tr>
<td>1L</td>
<td>2</td>
<td>GT</td>
<td>FIU...</td>
<td>fdx ilh ilhp</td>
<td>jump if greater than</td>
</tr>
<tr>
<td>1L</td>
<td>2</td>
<td>LE</td>
<td>FIU...</td>
<td>fdx ilh ilhp</td>
<td>jump if less than or equal</td>
</tr>
<tr>
<td>1L</td>
<td>2</td>
<td>LT</td>
<td>FIU...</td>
<td>fdx ilh ilhp</td>
<td>jump if less than</td>
</tr>
<tr>
<td>1L</td>
<td>2</td>
<td>NE</td>
<td>FIU...</td>
<td>fdx ilh ilhp</td>
<td>jump if not equal</td>
</tr>
<tr>
<td>2S</td>
<td>1</td>
<td>ARG</td>
<td>FIUP.B</td>
<td>fdx ilh p</td>
<td>argument</td>
</tr>
<tr>
<td>1T</td>
<td>1</td>
<td>CALL</td>
<td>FIUPV...</td>
<td>fdx ilh p</td>
<td>function call</td>
</tr>
<tr>
<td>1T</td>
<td>2</td>
<td>CALL</td>
<td>...B</td>
<td></td>
<td>function return</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>JUMP</td>
<td>....V.</td>
<td></td>
<td>unconditional jump</td>
</tr>
<tr>
<td>1L</td>
<td>0</td>
<td>LABEL</td>
<td>....V.</td>
<td></td>
<td>label definition</td>
</tr>
</tbody>
</table>
into sets for each type suffix, except that a single set is used for both I and U when the set for I is identical to the set for U; INDIR is an example.

The actual values for the size indicators, \texttt{fdcsilhp}, depend on the target. A specification like ADDF\texttt{f} denotes the operator ADD+sizeop(\texttt{f}), where \texttt{f} is replaced by a target-dependent value, e.g., ADDF4 and ADDF8. For example, back ends must implement the following CVI and MUL operators.

\begin{verbatim}
CVIF/CVIFd CVIFx
CVII/c CVII/s CVII/i CVII/l CVII/h
CVIU/c CVIU/s CVIU/i CVIU/l CVIU/h CVIU/p

MULF/MULFd MULFx
MULI/MULI/l MULI/h
MULU/MULU/l MULU/h
\end{verbatim}

On most platforms, there are fewer than three sizes of floats and six sizes of integers, and pointers are usually the same size as one of the integers. And lcc doesn't support the 'long long' type, so \texttt{h} is not currently used. So, the set of platform-specific operators is usually smaller than the list above suggests. For example, the X86, SPARC, and MIPS back ends implement the following CVI and MUL operators.

\begin{verbatim}
CVIF4 CVIF8
CVII1 CVII2 CVII4
CVIU1 CVIU2 CVIU4

MULF4 MULF8
MULI4
MULU4
\end{verbatim}

The set of operators is thus target-dependent; for example, ADDI8 appears only if the target supports an 8-byte integer type. The lcc distribution includes a program, \texttt{ops.c}, that, given a set of sizes, prints the required operators and their values, e.g.,

\begin{verbatim}
% ops c=1 s=2 i=4 l=4 h=4 f=4 d=8 x=8 p=4
... CVIF4=4225 CVIF8=8321
CVII1=1157 CVII2=2181 CVII4=4229
CVIU1=1158 CVIU2=2182 CVIU4=4230
... MULF4=4561 MULF8=8657
MULI4=4565
MULU4=4566
... 131 operators
\end{verbatim}

The leaf operators yield the address of a variable or the value of a constant. \texttt{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. The binary operators accept two numbers and yield one.

The type suffix for a conversion operator denotes the type of the result and the size indicator gives the size of the result. For example, CVUI4 converts an unsigned (U) to a 4-byte signed inte-
ger (I4). The syms[0] field points to a symbol-table entry for a integer constant that gives the size of the source operand. For example, if syms[0] in a CVUI4 points to a symbol-table entry for 2, the conversion widens a 2-byte unsigned integer to a 4-byte signed integer. Conversions that widen unsigned integers zero-extend; those that widen signed integers sign-extend.

The front end composes conversions between types $T_1$ and $T_2$ by widening $T_1$ to it's 'supertype', if necessary, converting that result to $T_2$'s supertype, then narrowing the result to $T_2$, if necessary. The following table lists the supertypes; omitted entries are their own supertypes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Supertype</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed char</td>
<td>int</td>
</tr>
<tr>
<td>signed short</td>
<td>int</td>
</tr>
<tr>
<td>unsigned char</td>
<td>int, if sizeof(char) &lt; sizeof(int) unsigned, otherwise</td>
</tr>
<tr>
<td>unsigned short</td>
<td>int, if sizeof(short) &lt; sizeof(int) unsigned, otherwise</td>
</tr>
<tr>
<td>void *</td>
<td>an unsigned type as large as a pointer</td>
</tr>
</tbody>
</table>

Pointers are converted to an unsigned type of the same size, even when that type is not one of the integer types.

For example, the front end converts a signed short to a float by first converting it to an int and then to a float. It converts an unsigned short to an int with a single CVUIi conversion, when shorts are smaller than ints.

ASGN stores the value of kids[1] into 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. These are most useful for ASGNNB, which assigns structures and initializes automatic arrays.

JUMP is an unconditional jump to the address computed by kids[0]. For most jumps, kids[0] is a constant ADDR node, but switch statements compute a variable target, so kids[0] can be an arbitrary computation. LABEL defines the label given by syms[0], and is otherwise a no-op. For the comparisons, syms[0] points to a symbol-table entry for the label to jump to if the comparison is true.

Function calls have a CALL node preceded by zero or more ARG nodes. The front end unnests function calls—it performs the inner call first, assigns its value to a temporary, and uses the temporary henceforth—so ARG nodes are always associated with the next CALL node in the forest. If wants_dag is one, CALL nodes always appear as roots in the forest. If wants_dag is zero, only CALL+V nodes appear as roots; other CALL nodes appear as right operands to ASGN nodes, which are roots.

A CALL node's syms[0] points to a symbol whose only nonnull field is type, which is the function type of the callee.

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 and alignment of the argument.

In CALL nodes, kids[0] computes the address of the callee. CALL+B nodes are used for calls to functions that return structures; kids[1] computes the address of a temporary local variable to hold the returned value. The CALL+B code and the function prologue must collaborate to store the CALL+B's kids[1] into the callee's first local. CALL+B nodes have a count of zero because the front end references the temporary wherever the returned value is referenced. There is no RET+B; the front end uses an ASGN+B to the structure addressed by the first local. CALL+B nodes appear only if the interface flag wants_callb is one; see Sec. 6. In RET nodes, kids[0] computes the value returned.
Character and short-integer actual arguments are always promoted to the corresponding integer type even in the presence of a prototype, because most machines must pass at least integers as arguments. Upon entry to the function, the promoted values are converted back to the type declared for the formal parameter. For example, the body of

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

becomes the two forests shown in Figure 4 on a 32-bit target. The solid lines are kids pointers and the dashed line is the link pointer. The left forest holds one dag, which narrows the widened actual argument to the type of the formal parameter. In the left dag, the left ADDRFP4 c refers to the formal parameter, and the one under the INDIRI4 refers to the actual argument. The right forest holds two dags. The first widens the formal parameter c to pass it as an integer, and the second calls f.

![Figure 4. Forests for \( f(\text{char } c) \{ f(c); \} \)](image)

In Table 1, the operators listed at and following ASGN are used for their side effects. They appear as roots in the forest, and their reference counts are zero. CALL+F, CALL+I, CALL+U, CALL+P also yield a value, in which case they appear as the right-hand side of an ASGN node and have a reference count of one. With this lone exception, all operators with side effects always appear as roots in the forest of dags, and they appear in the order in which they must be executed. The front end communicates all constraints on evaluation order by ordering the dags in the forest. If the Standard specifies that \( x \) must be evaluated before \( y \), then \( x \)'s dag will appear in the forest before \( y \)'s, or they will appear in the same dag with \( x \) in the subtree rooted by \( y \). An example is

\[ \text{int } i, \ *p; f() \{ i = *p++; \} \]

The code for the body of \( f \) generates the forest shown in Figure 5 (for a 32-bit target). The INDIRP4 fetches the value of \( p \), and the ASGNI4 changes \( p \)'s value to the sum computed by this INDIRP4 and 4. The ASGNI4 sets \( i \) to the integer pointed to by the original value of \( p \). Since the INDIRP4 appears in the forest before \( p \) is changed, the INDIRI4 is guaranteed to use the original value of \( p \).
5.1 Node Constructors

Node newnode(int op, Node left, Node right, Symbol sym)
  allocates a dag node; initializes the op field to op, kids[0] to left, kids[1] to right, and
  syms[0] to sym; and returns a pointer to the new node.

5.2 Operations on Node Operators

int generic(int op)
  return the generic operator for op; that is, an operator without a type suffix or size modifier.
  For example, generic(ADD+F+sizeop(8)) is ADD.

int opindex(int op)
  returns the operator number for op and is used to map the generic operators into a contiguous
  range of integers.

int opsize(int op)
  extracts and returns the size modifier from op. For example, opsize(ADD+F+sizeop(8)) is 8.

int optype(int op)
  returns the type suffix for op, which is one of F, I, U, P, V, or B. For example,
  optype(ADD+F+sizeop(8)) is F.

int sizeop(int n)
  converts n to a size modifier, which can be added to a generic operator and a type suffix; e.g.,
  ADD+F+sizeop(8) forms ADDF4, 4-byte floating-point addition.

int specific(int op)
  returns the type-specific operator for op; that is, an operator with a size modifier. For exam-
  ple, specific(ADD+F+sizeop(8)) is ADD+F.

6. Interface Flags

The interface flags help configure the front end for a target.

unsigned little_endian:1
  should be one if the target is a little endian and zero if it's a big endian. A computer is a little
  endian if the least significant byte in each word has the smallest address of the bytes in the
  word. For example, little endians lay out the word with the 32-bit unsigned value 0xAABBCCDD
  thus:

  AA BB CC DD
where the addresses of the bytes increase from the right to the left. A computer is a big endian if the least significant byte in each word has the largest address of the bytes in the word. For example, big endians lay out the word with the unsigned value 0xAABBCCDD thus:

```
DD CC BB AA
```

In other words, icc's front end lays out a list of bit fields in the addressing order of the bytes in an unsigned integer: from the least significant byte to the most significant byte on little endians and vice versa on big endians. The Standard permits either order, but following increasing addresses is the prevailing convention.

unsigned mulops_calls:1
should be zero if the hardware implements multiply, divide, and remainder. It should be one if the hardware leaves these operations to library routines. The front end unnests nested calls, so it needs to know which operators are emulated by calls.

unsigned wants_callb:1
tells the front end to emit CALL+B nodes to invoke functions that return structures. If wants_callb is zero, the front end generates no CALL+B nodes but implements them itself, using simpler operations: It passes an extra, leading, hidden argument that points to a temporary; it ends each structure function with an ASGN+B dag that copies the return value to this temporary; and it has the caller use this temporary when it needs the structure returned. When wants_callb is one, the front end generates CALL+B nodes. The kids[1] field of a CALL+B computes the address of the location at which to store the return value, and the first local of any function that returns a structure is assumed to hold this address. Back ends that set wants_callb to one must implement this convention by, for example, initializing the address of the first local accordingly. If wants_callb is zero, the back end cannot control the code for functions that return structure arguments, so it cannot, in general, mimic an existing calling convention.

unsigned wants_argb:1
tells the front end to emit ARG+B nodes to pass structure arguments. If wants_argb is zero, the front end generates no ARG+B nodes but implements structure arguments itself using simpler operations: It builds an ASGN+B dag that copies the structure argument to a temporary; it passes a pointer to the temporary; it adds an extra indirection to references to the parameter in the callee; and it changes the types of the callee's formals to reflect this convention. It also sets structarg for these parameters to distinguish them from bona fide structure pointers. If wants_argb is zero, the back end cannot control the code for structure arguments, so it cannot, in general, mimic an existing calling convention.

unsigned left_to_right:1
tells the front end to evaluate and to present the arguments to the back end left to right. That is, the ARG nodes that precede the CALL appear in the same order as the arguments in the source code. If left_to_right is zero, arguments are evaluated and presented right to left. The Standard permits either order.

unsigned wants_dag:1
tells the front end to pass dags to the back end. If it's zero, the front end undags all nodes with reference counts exceeding one. It creates a temporary, assigns the node to the temporary, and uses the temporary wherever the node had been used. When wants_dag is zero, all reference counts are thus zero or one, and only trees, which are degenerate dags, remain; there are no general dags.

unsigned unsigned_char:1
tells the front end whether plain characters are signed or unsigned. If it's zero, char is a signed type; otherwise, char is an unsigned type.


All the interface flags can be set by command-line options, e.g., `-Wf-unsigned_char=1` causes plain characters to be unsigned.

7. Initialization

During initialization, the front end calls

```c
void (*progbeg)(int argc, char *argv[])
```

`argv[0..argc-1]` point to those program arguments that are not recognized by the front end, and are thus deemed target-specific. `progbeg` processes such options and initializes the back end.

At the end of compilation, the front end calls

```c
void (*progend)(void)
```

to give the back end an opportunity to finalize its output. On some targets, `progend` has nothing to do and is thus empty.

8. Definitions

Whenever the front end defines a new symbol with `scope` `CONSTANTS`, `LABELS`, or `GLOBAL`, or a static variable, it calls

```c
void (*defsymbol)(Symbol p)
```

to give the back end an opportunity to initialize its `Xsymbol` field, `p->x`. For example, the back end might want to use a different name for the symbol. The conventions on some targets in this book prefix an underscore to global names. The `Xsymbol` fields of symbols with `scope` `PARAM` are initialized by `function`, those with `scope` `LOCAL+k` by `local`, and those that represent address computations by `address`.

A symbol is `exported` if it's defined in the module at hand and used in other modules. It's `imported` if it's used in the module at hand and defined in some other module. The front end calls

```c
void (*export)(Symbol p)
void (*import)(Symbol p)
```

to announce an exported or imported symbol. Only nonstatic variables and functions can be exported. The front end always calls `export` `before` defining the symbol, but it may call `import` at any time, before or after the symbol is used. Most targets require `export` to emit an assembler directive. Some require nothing from `import`.

```c
void (*global)(Symbol p)
```

emits code to define a global variable. The front end will already have called `segment`, described below, to direct the definition to the appropriate logical segment, and it will have set `p->u.seg` to that segment. It will follow the call to `global` with any appropriate calls to the data initialization functions. `global` must emit the necessary alignment directives and define the label.

The front end announces local variables by calling
void (*local)(Symbol p)

It announces temporaries likewise; these have the `p->temporary` flag set, which indicates that the symbol will be used only in the next call to `gen`. If a temporary's `p->u.t.cse` field is non-null, it points to the Node that computes the value assigned to the temporary. `local` must initialize `p->x`, which holds data like the local’s stack offset or register number.

The front end calls

```c
void (*address)(Symbol p, Symbol q, long n)
```

to initialize `q` 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 it does so based on the values of `p->x` and `n`. A typical `address` adds `p`'s stack offset to `n` for locals and parameters, and sets `q->x.name` to `p->x.name` concatenated with `+n` or `-n` for other variables. For example, if `n` is 40 and `p` points to a symbol with the source name `array`, and if the back end forms names by prefixing an underscore, then `address` will create the name `_array+40`, so that the addition can be done by the assembler instead of at run time. `address` accepts globals, parameters, and locals, and is called only after these symbols have been initialized by `defsymbol`, `function`, or `local`. Also, `address` is optional: If the function pointer in the interface record is null, the front end will not call `address`.

When the front end announces a symbol by calling one of the interface procedures above, it sets the symbol's `defined` flag after the call. This flag prevents the front end from announcing a symbol more than once.

Lcc's front end manages four logical segments that separate code, data, and literals:

```c
enum { CODE=1, BSS, DATA, LIT }
```

The front end emits executable code into the `CODE` segment, defines uninitialized variables in the `BSS` segment, and it defines and initializes initialized variables in the `DATA` segment and constants in the `LIT` segment. The front end calls

```c
void (*segment)(int seg)
```

to announce a segment change. The argument is one of the segment codes above. `segment` maps the logical segments onto the segments provided by the target machine.

`CODE` and `LIT` can be mapped to read-only segments; `BSS` and `DATA` must be mapped to segments that can be read and written. The `CODE` and `LIT` segments can be mapped to the same segment and thus combined. Any combination of `BSS`, `DATA`, and `LIT` can be combined likewise. `CODE` would be combined with them only on single-segment targets.

**9. Constants**

The interface functions

```c
void (*defaddress)(Symbol p)
void (*defconst)(int suffix, int size, Value v)
```
initialize constants. `defconst` emits directives to define a cell and initialize it to a constant value. `v` is the value, and `suffix` encodes its type and thus which element of the `Value v` to access, as shown in the following table.

<table>
<thead>
<tr>
<th>suffix</th>
<th>v Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>v.d</td>
<td>float, double, long double</td>
</tr>
<tr>
<td>I</td>
<td>v.i</td>
<td>signed char, signed short, signed int,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signed long</td>
</tr>
<tr>
<td>U</td>
<td>v.u</td>
<td>unsigned char, unsigned short,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned int, unsigned long</td>
</tr>
<tr>
<td>P</td>
<td>v.p</td>
<td>void *, function pointers</td>
</tr>
</tbody>
</table>

`defconst` must narrow `v.x` when `size` is less than `sizeof v.x`; e.g., to emit an unsigned char, `defconst` should emit `(unsigned char)v.u`.

If `suffix` is `P`, `v.p` holds a numeric constant of some pointer type. These originate in declarations like `char *p=(char*)0xF0`. `defaddress` initializes pointer constants that involve symbols instead of numbers.

The `defconst` functions in distributed back ends permit cross-compilation, so they compensate for different representations and byte orders. For example, they swap the two halves of a `double` if compiling for a big endian on a little endian or vice versa.

In general, Standard C compilers can't leave the encoding of floating-point constants to the assembler, because few assemblers implement C's casts. For example, the correct initialization for

```c
double x = (float)0.3
```

has zeros in the least significant bits. Typical assembler directives like

```assembly
.double 0.3
```

can't implement casts and thus erroneously initialize `x` without zeros in the least significant bits, so most `defconsts` must initialize doubles by emitting two unsigneds, for example.

```c
void (*defstring)(int n, char *s)
```

emits code to initialize a string of length `len` to the characters in `s`. The front end converts escape sequences like `\000` into the corresponding ASCII characters. Null bytes can be embedded in `s`, so they can't flag its end, which is why `defstring` accepts not just `s` but also its length.

```c
void (*space)(int n)
```

emits code to set aside `n` zero bytes.

**10. Functions**

The front end compiles functions into private data structures. It 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.
Three interface functions and two front-end functions collaborate to compile a function:

```c
void (*function)(Symbol f, Symbol caller[], Symbol callee[], int ncalls)
void (*emit)(Node)
Node (*gen)(Node)

void emitcode(void)
void gencode(Symbol caller[], Symbol callee[])
```

At the end of each function, the front end 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 procedure that traverses the front end's private structures and passes each forest of dags to the back end's gen, which selects code, annotates the dag to record its selection, and returns a dag pointer. gencode also calls local to announce new locals, block-beg and blockend to announce the beginning and end of each block, and so on. emitcode is a front-end procedure that traverses the private structures again and passes each of the pointers from gen to emit to emit 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 and does any other necessary per-function initializations. After calling gencode, the size of the procedure activation record, or frame, and the registers that need saving 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 points to the symbol for the current function, and ncalls is the number of calls to other functions made by the current function. ncalls helps on targets where leaf functions—those that make no calls—get special treatment.

The arguments caller and callee are arrays of pointers to symbols; a null pointer terminates each. 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 many functions, the symbols in each array are the same, but they can differ in both sclass and type. For example, in

```c
single(x) float x; { ... }
```

a call to single passes the actual argument as a double, but x is a float within single. Thus, caller[0]->type is doubletype, the front-end global that represents doubles, and callee[0]->type is floattype. And in

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

caller[0]->sclass is AUTO and callee[0]->sclass is REGISTER. Even without register declarations, the front end assigns frequently referenced parameters to the REGISTER class, and sets
callee's sclass} accordingly. To avoid thwarting the programmer's intentions, this assignment is made only when there are no explicit register locals.

caller and callee are passed to gencode. If caller[i]->type differs from callee[i]->type, or the value of caller[i]->sclass differs from callee[i]->sclass, 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 single 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 sclass to AUTO to prevent an unnecessary assignment of caller[i] to callee[i].

function could also change the value of caller[i]->sclass from AUTO to REGISTER if it wished to assign a register to that parameter. The MIPS calling convention, for example, passes some arguments in registers, so function assigns those registers to the corresponding callee in leaf functions. If, however, caller[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.

Most back ends define for each function activation an argument-build area to store the arguments to outgoing calls. The front end unnests calls, so the argument-build area can be used for all calls. The back end makes the area big enough to hold the largest argument list. When a function is called, the caller's argument-build area becomes the callee's actual arguments.

Calls are unnested because some targets pass some arguments in registers. If we try to generate code for a nested call like f(a,g(b)), and if arguments are evaluated and established left to right, it is hard not to generate code that loads a into the first argument register and then destroys it by loading b into the same register, because both a and b belong in the first argument register, but a belongs there later.

Some calling conventions push arguments on a stack. They can handle nested calls, so an argument-build area is not always necessary. Unnesting has the advantage that stack overflow can occur only at function entry, which is useful on targets that require explicit prologue code to detect stack overflow.

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 it estimates to be most frequently used. It assigns REGISTER class to even these locals if their addresses are not taken and if they are estimated to be used more than twice. This announcement order and sclass override collaborate to put the most promising locals in registers even if no registers were declared.

If p->sclass is REGISTER, local may decline to allocate a register and may change sclass to AUTO. The back end has no alternative if it has already assigned all available registers to more promising locals. As with parameters, local could assign a register to a local with sclass equal to AUTO and change sclass to REGISTER}, but it can do so only if the symbol's addressed flag is zero.

Source-language blocks bracket the lifetime of locals. gencode announces the beginning and end of a block by calling:

```c
void (*blockbeg)(Env *e)
void (*blockend)(Env *e)
```

Env, defined in config.h, is target-specific. It 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. For example, blockbeg typically records in *e the size of the frame and the registers that are busy at the beginning of the block, and blockend restores the register state and updates the stack if the new block has pushed deeper than the maximum depth seen so far.
The front end calls `gen` to select code. It passes `gen` a forest of dags. For example, Figure 5 shows the forest for

```c
int i, *p; f() { i = *p++; }
```

A postorder traversal of this forest yields the linearized representation shown in the table below.

<table>
<thead>
<tr>
<th>Node #</th>
<th>op</th>
<th>count</th>
<th>kids</th>
<th>syms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADDGRP4</td>
<td>2</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>INDIRP4</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>CNSTI4</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ADDP4</td>
<td>1</td>
<td>2, 3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ASGNP4</td>
<td>0</td>
<td>1, 4</td>
<td>4, 4</td>
</tr>
<tr>
<td>6</td>
<td>ADDRGP4</td>
<td>1</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>INDIRI4</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ASGNI4</td>
<td>0</td>
<td>6, 7</td>
<td>4, 4</td>
</tr>
</tbody>
</table>

This forest consists of three dags, rooted at nodes 2, 5, and 8. The INDIRP4 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. So `gen` merely annotates the nodes in their x fields to identify the code selected, and returns a pointer that is ultimately passed to the back end’s `emit` to 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.

`emit` emits a forest. Typically, it traverses the forest and emits code by switching on the opcode or some related value stored in the node by `gen`.

11. Interface Binding

The compiler option `-target=name` identifies the desired target. The name-interface pairs for the available targets are stored in

```c
typedef struct binding {
  char *name;
  Interface *ir;
} Binding;

Binding bindings[]
```

The front end identifies the one in the `-target` and stores a pointer to its interface record in

```c
Interface *IR
```

Whenever the front end needs to call an interface function, or read a type metric or an interface flag, it uses `IR`.

Back ends must define and initialize `bindings`, which associates names and interface records. For example, the back ends in lcc distribution define `bindings` in `bind.c`. 
12. Upcalls

In addition to the functions defined in the previous sections, back ends calls the front-end functions listed below to perform output, allocate storage, generate labels, and manage strings.

```c
void *allocate(unsigned long n, unsigned arena)
永久性地分配n字节在arena，其中可以是PERM，FUNC或STMT且返回
第一个字节的指针。该空间必须是对机器最
要求类型的对齐。分配在PERM的在编译完成时释放；分配在
FUNC和STMT在编译函数和语句后被释放。
```

```c
void fprint(FILE *f, const char *fmt, ...)
void print(const char *fmt, ...)
```

print formated data to the file stream f (fprint) or to standard output (print). These functions are like Standard C’s fprintf and printf, but support only some of the standard conversion specifiers and do not support flags, precision, and field-width specifications. They support additional conversion specifiers useful for generating code. The conversion specifiers are listing ed Table 2.

<table>
<thead>
<tr>
<th>Specifiers</th>
<th>Argument Types</th>
<th>Action or Corresponding printf Specifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d %D</td>
<td>int, long</td>
<td>%d %ld</td>
</tr>
<tr>
<td>%u %U</td>
<td>unsigned, unsigned long</td>
<td>%u %lu</td>
</tr>
<tr>
<td>%o</td>
<td>unsigned</td>
<td>%o</td>
</tr>
<tr>
<td>%x %X</td>
<td>unsigned, unsigned long</td>
<td>%x %lx</td>
</tr>
<tr>
<td>%f %e %g</td>
<td>double</td>
<td>%f %e %g</td>
</tr>
<tr>
<td>%p</td>
<td>void*</td>
<td>%x, if the argument is null</td>
</tr>
<tr>
<td>%s</td>
<td>const char*</td>
<td>%s</td>
</tr>
<tr>
<td>%S</td>
<td>const char*, int</td>
<td>print a string of a specified length</td>
</tr>
<tr>
<td>%I</td>
<td>int</td>
<td>print spaces</td>
</tr>
<tr>
<td>%k</td>
<td>int</td>
<td>print an English for a token code</td>
</tr>
<tr>
<td>%t</td>
<td>Type</td>
<td>print an English for a Type</td>
</tr>
<tr>
<td>%w</td>
<td>Coordinate</td>
<td>print a source coordinate</td>
</tr>
</tbody>
</table>

```c
int genlabel(int n)
    增加生成的识别器计数器n并返回其旧值。
```

```c
int roundup(int n, int m)
    n向上取到下一个m的倍数，m必须是2的幂。
```

```c
const char *string(const char *str)
    返回stringn(str, strlen(str))。
```

```c
const char *stringd(long n)
    返回n的字符串表示；stringd安装返回的字符串到字符串表，
    如果必要。
```

```c
const char *stringf(const char *fmt, ...)
    格式化其参数到一个字符串，安装该字符串到字符串表，如果必要，
    并返回一个指向安装的字符串的指针。See print for formatting details.
```
const char *stringn(const char *str, int len)
installs str[0..len-1] in the string table, if necessary, and returns a pointer to the installed copy. The string table holds only one copy of each unique string, so strings returned by any of the string functions can be compared for equality by comparing only their pointers.

References

1. C. W. Fraser and D. R. Hanson, A Retargetable Compiler for C: Design and Implementation, Addison-Wesley, Menlo Park, CA, 1995.