

Intermediate Representations

An Intermediate Representation (IR) is any internal or external representation of the source program which is complete and neither in source nor target form.

Early in the history of computing, compilers were generally designed as "one-pass" compilers, meaning they would emit the target code directly as the source language was recognized. This eliminated the need for large amounts of memory or disk space to store the IR. Many languages can be successfully compiled one-pass, including C (at least original C, not so much modern C).

However, such an approach results in very poor output code quality, as there is no opportunity to re-examine the code for optimizations. The output code follows the input code structure directly as it is parsed. One-pass compilers are limited to very constrained environments, e.g. a compiler which must run within an embedded device. All modern host-based compilers use one or more **intermediate representations (IR)** which progressively transform the source code to the target code.

The IR of the program is subjected to multiple optimization passes. The optimizers re-write the IR code. Depending on the design of the compiler, different IR forms may be used for different passes. After the last optimizer, the final IR is used as the input for the **target code generation** process.

There is no one "right answer" when designing an IR strategy. Some of the decisions the compiler writer will face hinge on the nature and extent of architecture-independent optimization which will take place after IR emission, on the nature and variety of back-ends which need to be supported, and the nature of the source language and the run-time environment.

Uses of IR

In some compilation frameworks, there is an externally visible form of IR, which has a standardized representation. But in others, the IR is strictly internal to the compiler. It exists in memory but is never written to disk, except for debugging purposes.

In traditional compilers such as gcc, the structure is frontend -> IR -> arch-neutral opt -> asssembly gen -> os-provided assembler -> os-provided linker -> a.out

This allows a cross-pluggable compiler, where different front ends (e.g. C, C++, Fortran) compile to a common IR, and the same optimizers can be used regardless of the source or the target languages.

In the LLVM (`http://www.llvm.org`) compilation framework, the IR is meant to be externalizable, with a `.llvm` filename extension. Thus llvm can replace the traditional object files (.o) and now the linker, instead of being the system linker (ld), is a linker which understands LLVM. An important benefit of this approach is it allows inter-procedural optimizations, also known as Link-Time Optimization (LTO). In this approach, the final llvm code is then assembled, linked with any non-llvm object files, and produces an a.out file.

In many systems, the IR is executed directly by an IR interpreter and run-time environment. Java is a popular example of a language which is compiled to an externally standardized IR known as Java Virtual Machine (JVM) bytecode. The bytecode can be executed on any system, regardless of native architecture.

Another possibility with external IR is so-called Just In Time (JIT) compilation, which is really a misnomer. It is, in fact, a deferred target code generation, in which the architecture-neutral IR is "compiled" to the target assembly/machine code when the program is executed. This allows the program to be shipped in semi-compiled form, retaining the cross-target portability, but still producing run-time execution speeds that are equal to traditionally compiled systems.

Popular interpreted languages such as Perl, PHP and Python are in fact semi-compiled to an IR which is then executed within the run-time framework. Here are some details: Python source code in `.py` files is first "compiled" into a bytecode representation which is stored in `.pyc` files. This happens the first time the script is invoked, or if the source code files have changed. Thereafter, the bytecode files are loaded directly, which speeds the script start-up process. The Python bytecode is a one-address (stack-based) architecture. The format of the `.pyc` bytecode is documented and is externalized, thus it satisfies our definition of an external, portable IR. This code is executed by the Python run-time engine. Because the compiler is also resident with the run-time engine, Python can act as a true interpreter and accept new python code as input data. PHP is designed to run primarily as an embedded interpreter in web server programs such as apache or nginx, although it can be run stand-alone. Its intermediate language is known as Zend and it is strictly internal. PHP also provides an API in which code written in C can be dynamically loaded into the PHP interpreter as it is running. This is known as "modules" and requires that all of the items of the PHP language also be visible as data structures and/or function calls from C. ZEND is a 3-address ("quads") based internal representation (see below). Each function or method is "compiled" into an array of quads. For example, these snippets from the PHP source code:

```
  from Zend/zend_language_parser.y:
expr:
        /*....*/
        |       expr '+' expr   { zend_do_binary_op(ZEND_ADD, &$$, &$1, &$3); }
        |       expr '-' expr   { zend_do_binary_op(ZEND_SUB, &$$, &$1, &$3); }
        |       expr '*' expr   { zend_do_binary_op(ZEND_MUL, &$$, &$1, &$3); }
        /*...*/

  from Zend/zend_compile.c:
void zend_do_binary_op(zend_uchar op, znode *result, znode *op1, znode *op2)
```

```
{
        zend_op *opline = get_next_op(CG(active_op_array) );

        opline->opcode = op;
        opline->result.op_type = IS_TMP_VAR;
        opline->result.u.var = get_temporary_variable(CG(active_op_array));
        opline->op1 = *op1;
        opline->op2 = *op2;
        *result = opline->result;
}
```

should be recognizable as quad generating (or will be, after reading this unit).

Perl is also a pure interpreter and is designed to work stand-alone, or be embedded into other programs. It uses an internal representation which can best be described as graphical. The complex AST data structure which is created during the parsing phase remains resident and is executed directly. Perl is also designed to be interfaced with C and publishes an API for its internals. Newer versions of Perl have support for dumping this internal data structure into an external "bytecode" form, (which is somewhat of a misnomer, as it is more of memory dump of the AST) and using that to speed up the initial script loading and parsing time.

IR Forms

The two basic forms of IR are graphical vs linear. A graphical IR is a directed graph data structure, i.e. one with a specific root representing the entire program (or some portion thereof), nodes representing abstract operations, and edges (pointers) linking nodes and values. A tangible example of a graphical IR is the Abstract Syntax Tree representation which was demonstrated in class and in previous units for encoding type information. In contrast, a linear IR more closely resembles assembly language, in that it is a simple list of operations.

In real-world compilers, neither form is used exclusively. It is common to find graphical IRs such as ASTs to represent types and expressions, especially in the earlier stages of compilation, and a series of linear IRs which make progress towards the final target code. Often, these linear IRs are augmented by graphs which track information such as data and control flow.

Almost all popular external IR forms are linear, because of the difficulty of concisely encoding and representing graphs externally.

Forms of Linear IR

A linear IR can be thought of as a sequence of assembly language instructions in a fairly abstract machine. This machine tends to be more generic and more powerful than an actual processor. During optimization, higher-level IR forms tend to be more expressive and allow the optimizer to gain a better understanding of the code and make better optimizations. On the other hand, the IR can not be too complex, otherwise it will be difficult to generate efficient target assembly code from it. Striking a balance is not an exact science.

Stack-based IR

One popular linear IR is **one-address code**, also known as **stack machine code**. It represents an abstract virtual machine in which there is a single stack of virtually unlimited size. The elements of the stack may be of a single, simple type (e.g. 8 byte integers) or they may be polymorphic (each stack element has both a type and a value, the representation and bounds of which depend on the type) Each operation of the one-address linear IR contains an operation code and a single optional parameter (hence the name). All data flow takes place through the stack. In addition to the stack, there has to be a way to refer to the instructions (a virtual program counter) so that branches can occur, and it is useful to have a notion of named variables (although lacking that all variables could be allocated from the stack at start-up).

The primary advantage of using a one-address code is simplicity. Not having to worry about intermediate values and register allocation makes it easy to generate one-address code during parsing, in fact it lends itself well to simple, one-pass compilation. The one-address code is also easy to write an interpreter for, making it a good choice for architecture- independent bytecode forms and for small, embedded compiler/interpreters. One-address code does not lend itself readily to code generation for most modern processors. That being said, a lot of work has been done recently on one-address codes (stack-based machines) because both Java and Python use this approach. Let's see how a compiler might generate one-address code for a simple expression:

```
f()
{
 extern int x,y,z;

          x=y*3+z*2;
}

ONE-ADDRESS CODE GENERATED FOR EXPRESSION ABOVE:

1         LOAD                Y
2         PUSH                3
3         MUL
4         LOAD                Z
5         PUSH                2
6         MUL
7         ADD
8         DUP
9         STORE               X
```

It would be fairly easy to implement this inside the parser:

```
%left '*' '/'
```

```
%left '+' '-'
%right '='
%%
expr:
          IDENT               {emit("LOAD",$1);}
          |NUMBER             {emit("PUSH",$1);}
          |expr '+' expr      {emit("ADD");}
          |expr '-' expr      {emit("SUB");}
          |expr '*' expr      {emit("MUL");}
          |expr '/' expr      {emit("DIV");}
          |IDENT '=' expr     {emit("DUP");emit("STORE",$1);}
          ;
```

By structuring the yacc/bison grammar to match the correct order of evaluation (including the use of the yacc/bison precedence rules mechanism), the one-address code will be emitted in the correct sequence to effect the evaluation of the expression. Comparing this code to the calculator example given in Unit 2, one would see that instead of evaluating the expression directly in the semantic actions, we are emitting the one-address code instead. The run-time stack of the one-address virtual machine then serves the same function as the parse-time yacc semantic value stack would for an interpreter/calculator (such as our example in unit #2).

One-address code can trivally be converted to three-address code (described immediately below) by considering each position on the (potentially boundless) stack to be a distinct temporary variable.

Three-Address Code / Quads

The most popular form of linear IR is **three-address code**. Each instruction has an operation code, up to two optional source parameters, and a destination parameter. E.g. A=B+C is in three-address form: the operation is addition, the source parameters are B and C, and the destination is A. Because each operation has four fields, three-address code is also called **quads**. Note that A,B and C need not be different operands, and that the order of operations is defined such that the source parameters B and C are read first, then the destination is written with the result. Therefore A=A+A works as expected.

Many processors actually implement a one-address or two-address system. An accumulator-centric processor is a one-address system. It requires most operations to go through an accumulator register, which acts as the destination and one of the source parameters, the other source parameter (if needed) being supplied as part of the instruction. Accumulator-based architectures are common on small, embedded processors.

Two-address instruction sets typically have a source and destination field, with the destination also being one of the source operands. Some architectures allow either field to be either a register or a memory location, while some impose further restrictions. X86 is an example of a two-address architecture.

Three-address architectures are more typically associated with RISC processors, such as SPARC or MIPS.

Regardless of the eventual target architecture, using a three-address IR form is the best idea. It allows the greatest ease in IR generation and the greatest clarity. Later, during actual target code generation, the restrictions of the specific architecture can always be applied to the more generic three-address code. Going in the other direction would be very difficult: if the IR were in two-address or one-address form, it would be difficult for the code generator to see opportunities for applying three-address form if that were the supported architecture.

Representing Quads

In the field of compiler theory, various notations are used to describe quads in human-readable form, e.g.:

```
res = OP src1,src2
res = src1 OP src2
res <- src1 OP src2
src1 OP src2 -> res
OP src1,src2,res
OP res,src1,src2
OP src1,src2 -> res
```

We'll generally stick to the first form, but do not be alarmed if other forms are seen in reading the various texts. If you understand what quads are supposed to do, it will quickly become apparent what the intention of the notation is.

Encoding Quads & Hidden Addressing Modes

At this stage of the compiler (quad generation), we can identify several "addressing modes", i.e. ways in which the source and result operands can be accessed:

• A local variable: At code generation time, local variables are typically given a slot in the function's stack frame. However, if there are sufficient registers, and the address of the local variable is never "exposed" by taking a pointer to it (in C. In languages that implement call-by-reference, passing the local variable to a function may also expose it), then the local variable may live entirely in a register and not "own" any memory.

In the examples of this unit, we will treat local variables as named objects that can be used directly in an operation.

• A function parameter: Function parameters behave like local variables as far as the C language definition is concerned, but at target code generation time, they may be accessed using different rules (e.g. some architectures pass parameters in specific registers). We will treat them as local variables

• A global variable: Access to global variables will eventually result in the use of an absolute address mode in the assembly language output. Global variables can not be assigned to a register, although they certainly can be loaded into a register for computation and the result stored from the register back into that variable. We will express global variables in the same way as local variables, and introduce a syntax to distinguish them.

• A numeric constant: Constants may appear explicitly in the source program, or may arise implicity (e.g. the multiplier used in pointer/array offset arithmetic). We'll have a bit more to say about **constant propagation** later on.

• An address constant: The assembler/linker conventions allow the use of a symbol which is offset by an integer. E.g.

```
extern char a;
int z=&a+4;
```

The compiler will pass through the symbolic expression "a+4" to the initializer which appears with symbol z in the assembly language output. This expression will be carried through by the linker, which will calculate the actual numeric value of the expression once linkage has been resolved and the symbol a has been assigned an absolute address.

If the intent of the IR is to make it externally transferrable, then addressing mode information must be externalized. In our example, the IR is intended for internal use only. We will therefore take advantage of the existence of the symbol table, and use a data structure such as:

```
struct quad {
        int opcode;
        union generic_node *result,*src1,*src2;
};
```

These generic_nodes can be allocated on-the-fly for temporary values and for constants (during optimization it is important to track constants too). Implicit in this encoding is that src1 and src2 are expected to fetch the actual value of the operand(s), while result must provide a means of storing the result. I.e. src1 and src2 are rvalues, while result is an lvalue.

Other Approaches to Addressing Modes and Types

The examples given in this unit are by no means the only way. Let's look at some of the important features of the LLVM IR:

• In LLVM, global variables are treated as constant pointers to the symbolic memory address which they will occupy at run-time. Global variables are accessed by assigning that symbolic constant to a temporary variable and using the LOAD/STORE mechanism, as if the variables were being accessed explicitly by a pointer.

• Variables which have been declared as local variables in LLVM are also accessed through a pointer. The memory address for the local variable comes from an LLVM built-in operator alloca, which is similar to the standard C library function, and has the

behavior of allocating a slice of memory from the stack frame.
• LLVM has a syntax to declare the names and types of function parameters. They can then be accessed directly.
• In LLVM, every operand is also flagged with a type. The notion of type can be complicated, including structure or array types.

Temporary Values

In constructing either target code or IR for expressions, there will naturally be intermediate, temporary values which arise as a result of having to evaluate each sub-expression as a binary or unary operation. These temporary values have no corresponding existence in the source program and therefore don't "own" a memory location as do variables.

When the compiler reaches the phase of generating target assembly code, it needs to have a way of managing these temporary values. They may be kept in registers, pushed and popped on the stack, or stored in the "stack frame" (more about that in a later unit) of the function as if they were local variables.

The trend in modern compiler design is to defer allocation of registers until fairly late in the game. Most IRs assume that there is an infinite supply of virtual registers, in which temporary variables can be placed. This simplifies the optimizer and allows it to perform a series of analyses based on data flow. The result is that many of the temporary variables may be eliminated. Then the register allocator re-writes the IR, replacing some of the variables (including possibly actual, declared local variables which are frequently used) with registers, and calculating how much "spill" area will be needed to hold temporary values if they don't all fit within the register pool.

For example, consider this code fragment:

```
int a,b,c,d;
          a=b*c+d*10;
```

Clearly, two temporary variables will be needed to hold the sub-expression results. But we could have also written:

```
int a,b,c,d,t1,t2;
          t1=b*c;
          t2=d*10;
          a=t1+t2;
```

In a good, optimizing compiler, both of these programs will result in the emission of the same target code (assuming that t1 and t2 are not used again in the program). This illustrates a principle of optimization: The optimizer should be able to see potential savings which the programmer did not explicitly code. Conversely, the programmer should be able to express code in the source language which best represents the problem at hand, and let the compiler worry about the implementation details.

Generating Quads for Expressions

Expressions in C are very powerful, and in fact include hidden flow-of-control constructs (e.g. the ?:, &&, || operators). We will consider simple expressions first, then examine control flow constructs, and then see how complicated operators are handled.

IR for expressions can be generated on-the-fly as a result of semantic actions. Because the grammar for expressions also specifies (loosely) the order of evaluation, we could embed actions to emit IR whenever an expression or sub-expression is reduced. However, in a non-trivial language such as C, it is often necessary to look at an expression in multiple ways. We might be trying to figure out the expression's lvalue, its rvalue, or its type (and therefore its size and alignment restrictions). It is therefore useful to have the parser construct an AST representation of each expression, and then re-traverse that AST at some later time (e.g. at the conclusion of each statement, or after the entire function has been converted to AST form). During AST construction, certain transformations or simplifications can be applied. There is no need to encode parentheses into the AST, for example, and compound assignment operators could be expanded out as if they were written in their equivalent form.

IR construction from an AST representation of an expression is a recursive process, using a depth-first (aka post-order) walk of the tree. We recurse on each of the operands, so the IR to compute them is taken care of. Then we examine the type of the operands, and generate code to perform any necessary conversions (e.g. when adding a double to an int, the int must be promoted). Finally, we emit the code to perform the operation for our node.

It will be to the compiler writer's advantage to make the quad opcodes powerful and generic. At the very least, all of the basic unary and binary operators should be modeled. It may be necessary to add an operation "type or width" to the opcode, e.g. to distinguish between int and long int arithmetic. The target code to be generated for `a*b` obviously involves a multiply instruction (unless the target machine lacks hardware multiply), but the exact instruction or sequence of instructions is different if a and b are longs versus if they are doubles. One could use a generic multiply IR opcode and then push the decision deeper into the bottom half, but it would be nice to stop worrying about arbitrary C language types after IR generation. Besides, the types are already being evaluated as we descend the expression AST.

In our examples, we will assume that all operations are in 32 bit integer arithmetic, and we will omit notation that distinguishes between the different types of variables. This will allow us to focus on the core issues of the IR generation without the visual clutter that will actually be present.

Consider the expression a=(b+c*10)*a, assuming a, b and c are local int variables. Clearly, there are intermediate results which do not correspond to any variable, and therefore temporary variables will be required. Here is a possible sequence of quads:

```
%T1=    MOV     b
%T2=    MOV     c
%T3=    MUL %T2,10
```

```
%T4=      ADD %T1,%T3
%T5=      MOV      a
%T6=      MUL %T4,%T5
a=        MOV      %T6
```

In the notation that will be used in these notes, temporary variables are denoted with a leading percent sign. This is the same character that identifies registers in the so-called "AT&T" or "UNIX" style of assembly language that we'll see in a subsequent unit. Temporary variables, we will see, are in effect virtual registers, so the use of % is justified. The `MOV` IR opcode is a simple assignment. Often these can be eliminated either by more optimal IR generation, or by subsequent optimization passes.

It should be apparent that many of the temporary variables and operations can be eliminated. Here is a possible optimized sequence:

```
%T1=      MUL c,10
%T2=      ADD b,%T1
a=        MUL %T2,a
```

We could choose to emit totally unoptimized IR (the first case) and let the optimizer fix it. Or we could exert more effort in creating a tighter IR. The trend in modern compilers is the former, since memory and CPU cycles are fairly cheap. This allows the compiler writer to focus on simplicity and correctness during IR generation.

In older compilers, or when writing a compiler that will compile in a more constrained environment such as a small embedded system, it might pay to generate better IR in the first pass, because our optimizer might be minimal or non-existent.

To generate the unoptimized IR code, we simply assign a new temporary variable for every subexpression. To generate the more optimal sequence, we need to, at each step of the recursion, be informed of the target for the value of our node. Schematically:

```
gen_rvalue(node,target)                //return value is the result name
                 //If target is NULL, we are expected to create a temporary
{
          if (node->type==SCALAR VARIABLE) return node;
          if (node->type==CONSTANT) return node;
          if (node->type is a BINARY OPERATOR)
          {
                    //Ordinary binary operator case, ignoring type conversions
                    left=gen_rvalue(node->left,NULL);
                    right=gen_rvalue(node->right,NULL);
                    if (!target) target=new_temporary();
                    emit(node->opcode,left,right,target);
                    return target;
          }
}
```

The `new_temporary()` function creates a new node of type `TEMPORARY`. We could attempt to optimize the allocation of temporary values, as if they were registers. In the example above, the target of the second quad could be T1, rather than T2, because T1 is not "live" after that second quad ("live" meaning that its value might be accessed again by another quad). However, this sort of live variable analysis is also something which the optimizer will be performing, not just on temporaries, but on all variables. So it will be easier to just assign a new, unique temporary number each time we need one, and let the optimizer figure it out later. The function `emit` creates a new quad with the given 4 arguments and appends it to the list of quads.

Pointers

What quads should we generate for the following code:

```
int *p,b,c;
        b= *p+1;
```

The value which is placed into variable b is the contents of the memory location (integer-sized) whose address is contained in the variable p. In order to represent this in a quad, we could introduce a new addressing mode, e.g. `b=ADD (*p),1`. But doing so would violate the purity which we had previously established. So we need to introduce a new quad:

```
%T1=    LOAD    p
b=      ADD     %T1,1
```

This may seem inefficient, in that it takes two quads instead of one to express the operation, but quads are not the target code. They are merely a tool for making progress towards the optimal target code. During target-specific instruction selection, a target code generator might recognize the template of a LOAD into a temp followed by an access to that same temp, and emit an appropriate instruction which makes use of an indirect addressing mode on the target computer.

We need to add some lines to our previous `gen_rvalue`:

```
gen_rvalue(node,target)                //return value is the result name
{
        /*...*/
        if (node->type == POINTER_DEREF)
        {
                addr=gen_rvalue(node->to,NULL);
                if (!target) target=new_temporary();
                emit(LOAD,addr,NULL,target);
                return target;
        }
        *...*/
```

Along similar lines of thought, let’s generate IR for `*p=b+c`

```
%T1=    ADD     b,c
        STORE   %T1,p
```

Here we have chosen to represent the memory address at which T1 is stored as the second source parameter to the STORE quad opcode, rather than as the destination (p=STORE %T1). This allows us to keep dataflow analysis clean, in that the STORE operation does not change the value of p. We'll see in later units that pointers greatly complicate dataflow analysis in general, for unless we have some way of narrowing down the objects to which p might be pointing, we must assume that any accessible variable could have its value changed as a result of that operation!

Assignments

We have already informally handled assignments where the lvalue is a simple variable. In general, creating the quads for an assignment expression involves two parts: Computing the value of the right side (the rvalue), and determining how to store that value on the left side (the lvalue).

```
union node *gen_lvalue(union node *node,int *mode)
{
        if (node->type == SCALAR VARIABLE) {*mode=DIRECT;return node}
        if (node->type == CONSTANT) return NULL;
        if (node->type == DEREF)  {
                *mode=INDIRECT;
                return=gen_rvalue(node->child,NULL);
        }
        //etc.
}
```

gen_lvalue can either return the place which is a direct lvalue, or it can return an rvalue which can be used indirectly as an lvalue through a pointer STORE operation. (It can also return an error if the operand can not be an lvalue, e.g. the expression 2=3) Given this, we can generate assignment:

```
union node *gen_assign(union node *node)
{
        dst=gen_lvalue(node->left,&dstmode);
        if (dst==NULL) {report error, invalid lhs of assignment}
        if (dstmode==DIRECT)
                gen_rvalue(node->right,dst);
        else
        {
                t1=gen_rvalue(node->right,NULL);
                emit(STORE,t1,dst,NULL);
        }
}
```

Note that in the skeletal code above, type conversions are not considered. It is assumed that the rhs and lhs of the assignment are identical types. If they are not, gen_assign would have to insert a type conversion operation. Another approach is to make a pass over the AST before IR generation and insert type cast nodes where needed, as if they had been explicitly placed in the expression (e.g. `int a;a=(int)1.0;`)

Type Conversions

Many type conversions, whether they are implicit from C's type promotion rules, or explicit casts, do not actually change the binary value of a variable and thus are a null operation from an IR standpoint. Where actual IR needs to be generated is when there is a change of representation, e.g. converting from a float to an int. When converting an unsigned short to a long, the most significant bits are 0-filled, but going from a signed short to a signed long, the sign bit of the source must be extended to fill the most significant bits of the destination.

It is best to come up with some generic conversion IR opcodes, e.g. CVTFI, and defer implementation to the instruction selection phase of final code generation.

Arrays and Pointer Arithmetic

Consider:

```
int a[10],v;
        v= *a;
```

Recall from Unit 4 that in an expression, something of type "array of X" is transformed to type "pointer to X" (except when that something is the direct argument of the address-of or sizeof operator). In generating the quads for the expression above, we need to obtain the address of the start of the array `a`, then access the memory at that address:

```
%T1=    LEA     a
%T2=    LOAD    %T1
v=      MOV     %T2
```

The `LEA` (Load Effective Address) quad obtains not the value of its operand, but its memory address. We are choosing not to expose at this time how the address is computed. E.g. if `a` is a local variable, then the address might be obtained in assembly language by adding an offset to the stack frame pointer register.

To modify our quad generation algorithm to be array-aware, we need to add a line to `gen_rvalue()`:

```
        /*...*/
        if (node->type is ARRAY VARIABLE)
        {
                temp=new_temporary();
                emit(LEA,node,NULL,temp);
                return temp;
        }
        /*...*/
```

Consider `a[i]=v`, where a is an array of ints, i and v are ints. We know from Unit 4 that this expression is equivalent to `*(a+i)=v`. (In fact, one potential approach to AST generation is to apply this equivalency, which is guaranteed by the C language standard, and dispense entirely with the [] operator as an AST node. This would simplify the number of cases that need to be coded both for AST and IR generation) Furthermore, we know that when pointers and ints are added, there is an implicit multiply by the sizeof the base type of the pointer. Therefore, the quads for this expression are:

```
%T1=     LEA      a
%T2=     MUL      i,4
%T3=     ADD      %T1,%T2
         STORE    v,%T3
```

We'll need to add more code to `gen_rvalue` to insert the MUL operation when performing addition of a pointer to an integer. Handling of subtracting an integer from a pointer is similar. We also need to consider subtraction of a pointer from a pointer, which creates a `DIV` operation. The implementation of these items is left as an exercise for the reader.

In generating quads as shown above for array reference, we have chosen to break this operation down into simpler steps. Some texts illustrate array references by introducing an explicit [] operator. Since in the C language, arrays are nothing more than pointer arithmetic, this would be of dubious value. Some target architectures (e.g. X86) might support an addressing mode specifically designed to streamline array access, which combines the multiplication and addition above. We have made the IR more primitive than the machine language, and it would be up to the back-end optimizer to recognize this multiply/add/indirect sequence and transform it into the appropriate addressing mode. In the LLVM IR, array types are represented fully, and an operator `getelementptr` exists to perform the array index computation. So the above example might be expressed as:

```
%T3=     GETELEM  a,i
         STORE    v,%T3
```

Multi-dimensional arrays

Now, consider a multi-dimensional array: `v=a[i][j]`, where a is declared `int a[10][20]`. We know from the definition of the C language that this is equivalent to `(a[i])[j]`, because C doesn't really support multi-dimensional arrays. Therefore, the minor subscript operation is higher in the AST. Further, this is equivalent to `*((*(a+i))+j)`. Let's begin to generate quads for this expression, starting with the innermost subexpression:

```
%T1=     LEA      a
%T2=     MUL      i,80
%T3=     ADD      %T1,%T2
```

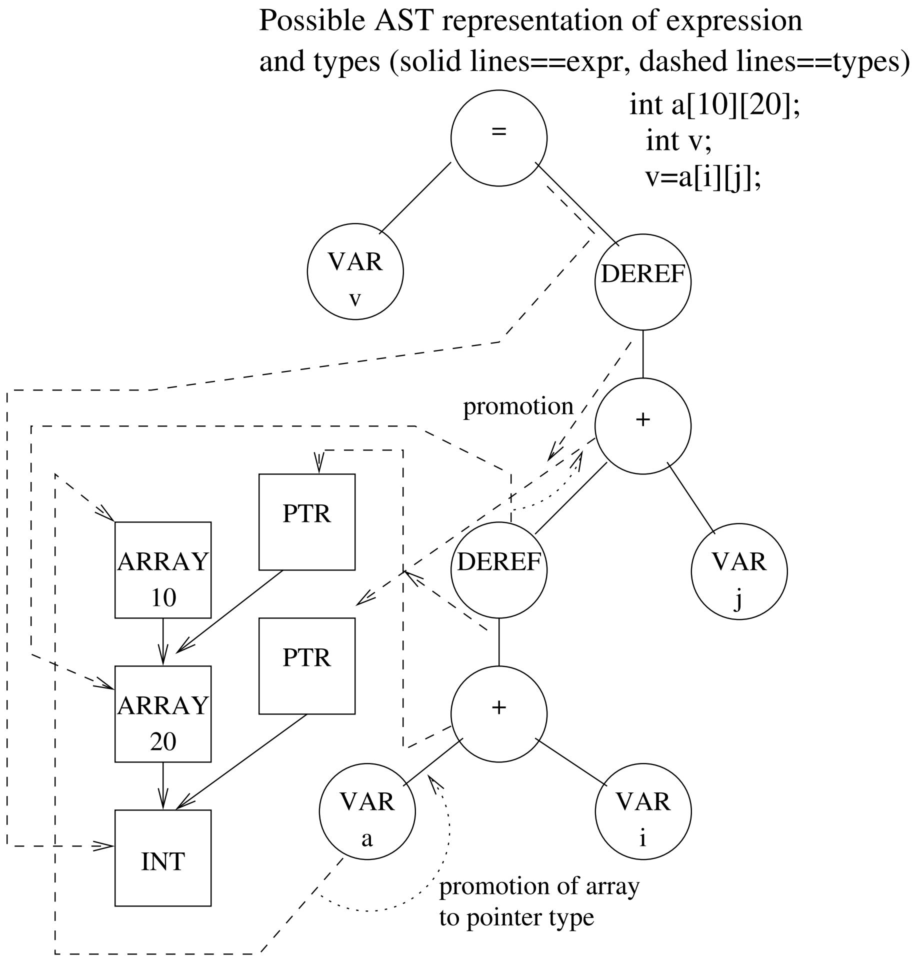
Temporary T3 will contain the value of a+i. In this subexpression, a is of type array_of(10,array_of(20,ints)) which is instantly promoted to pointer_to (array_of(20,ints)). Therefore, the pointer arithmetic multiplier is 80 (20*sizeof(int)). The next innermost subexpression is *(a+i) i.e. *T3. In the earlier examples of dereferencing a pointer, we emitted a LOAD instruction to fetch the rvalue. But here we have a pointer to an array type. The rvalue of an array is simply its address, and T3 is already pointing at the array and thus contains its address, so the dereference operation is a null operation, and we do not need to emit any additional quads. Returning to our work-in-progress gen_rvalue function, we need to modify it slightly:

```
gen_rvalue(node,target)                //return value is the result name
{
          /*...*/
          if (node->type == POINTER_DEREF)
          {
                    if (node->to is of ARRAY type)
                    {
                              return gen_rvalue(node->to,target);
                    }
                    addr=gen_rvalue(node->to,NULL);
                    if (!target) target=new_temporary();
                    emit(LOAD,addr,NULL,target);
                    return target;
          }
          *...*/
```

Now let us complete the quad generation by emitting code for the outermost addition and dereferencing operations:

```
%T4=      MUL       j,4
%T5=      ADD       %T3,%T4
v=        LOAD      %T5
```

The illustration below shows the AST structure of the expression, an AST representation of the type of multi-dimensional array variable a, and the type promotions and transformations which take place.



Structures

A structure is a bunch of bytes of known size, containing within it structure members at specific offsets. When a (complete) structure definition is processed, the layout of the structure can be determined. In order to do so, we must have knowledge of the target architecture, as we need to be able to compute the sizeof each member. We also need to know about possible alignment issues on that architecture. We will take up these issues when we discuss specific targets. For now, let us assume that the offsets of the structure members were determined at declaration time.

When we see an expression `s.e`, where `s` is of structure type, and `e` is a bare identifier, we can consult the symbol table and find the entry for member `e` in that structure type. This entry contains the offset in bytes of the member, as well as the type of the member. Consider this example:

```
struct s {int a,b,c;} *p,s;
```

```
int i;
        i=p->b;
        i=s.c;
```

We assume that a,b,c have been assigned offsets 0, 4 and 8. The IR which would be generated for the first assignment is:

```
%T1=    ADD             p,4
i=      LOAD            T1
```

Now, for the second assignment:

```
%T1=    LEA             s
%T2=    ADD             T1,8
i=      LOAD            T2
```

In order to access the rvalue of a direct selection expression, we need to, in fact, find the address of its right operand relative to the left operand. The reader may see an analogy here with array accesses. Let's see how we can add support for structures to our quad generator functions:

```
gen_rvalue(node,target)
{
        /*...*/
        if (node->type == DIRECT_SELECTION || node->type==INDIRECT_SELECTION)
        {
                if (node->type==DIRECT_SELECTION)
                        base=gen_addressof(node->left);
                else
                        base=gen_rvalue(node->left);
                sym_entry=symbol_lookup(node->left,node->right);
                temp=new_temporary();
                emit(ADD,base,sym_entry->offset,temp);
                if (!target) target=new_temporary();
                emit(LOAD,temp,NULL,target);
        }
```

`gen_addressof()` is a function similar to `gen_lvalue`. Depending on implementation, the two functions might be merged. `gen_addressof` is also the function we'd need to handle the `&` operator.

C allows for the assignment of structs, but not of arrays. This is fairly easily handled:

```
gen_assignment(node)
{
        /*...*/
        if (type_of(node->right) is STRUCT &&
                type_of(node->left) is STRUCT)
        {
                check that types are  identical
                emit(COPY,gen_addressof(node->left),
                        size_of(node->right),gen_addressof(node->right));
        }
```

Here we introduce an IR instruction dst=COPY src,sz in which src and dst are both interpreted as memory addresses, and sz is the number of bytes to copy. If the structure has fairly few members, it might be more efficient to generate quads which copy the structure member-by-member.

It is left as an exercise for the reader to explore how more complicated expressions involving arrays, pointers and structures would be handled.

An argument could be made for not breaking down the structure access mechanism in the IR, and instead introducing specific operations which mimic the . and -> operators in C. The Dragon text uses this approach. In the schema which we have been using, all quad operations involve primitive scalars, and there is no mechanism for representing a composite value.

Because LLVM has a rich type system, structures are carried as a composite type, and the member selection operator is carried through as a quad opcode, with the actual address computations pushed down to target code generation phase.

Unions

A union is a bunch of bytes whose size is large enough to hold the largest member of the union (including possible padding needed for alignment restrictions). From an assembly-language standpoint, the existence of a union is a "don't-care." Consider:

```
union u {
          int i;
          int a[10];
} u;

u.i++;
```

The access to the union member is basically equivalent to:

```
*(int *)(&u)++;
```

A cast which changes a pointer to X to a pointer to Y is a null operation, because (unless the architecture is such that these pointers have different representations, which is generally not the case) the value of the pointer itself will not change. Therefore, the quad for the example above is simply:

```
u=ADD     1,u
```

Constant Folding & Propagation

It is possible during IR generation to see certain opportunities for replacing expressions with constants. E.g.

```
a=3+4*2;

%T1=      MUL       4,2
a=        ADD       3,T1
```

Clearly this can be replaced with a simple `a=MOV 11`. In order to perform this **constant folding**, our `gen_rvalue` needs to be given the intelligence to recognize that an operation being considered can be evaluated entirely at compile-time, and return a CONSTANT node for that value. It may be the case that all of the operands of an AST node are CONSTANT, or an algebraic identity might be seen, e.g. a multiply by a constant 0.

We'll see during the optimization unit that much more sophisticated **constant propagation** is possible. E.g.:

```
a=2+2;
f();
b=a;
c=b+1;
```

Simple constant folding during AST traversal would eliminate the first computation of a, replacing it with a constant a=MOV 4. But whether or not the computation of c can be replaced with a c=MOV 5 can not be determined in this manner. We need to perform dataflow analysis to realize that the constant value which went into a in the first line then reached the 3rd and 4th lines. Complicating this analysis is the function call. If a is a variable which might possibly be exposed to function f(), and the code of f() is not known, then the function call invalidates any knowledge of a's value.

In general, both constant folding during AST traversal and more sophisticated constant propagation dataflow analysis are done. The former is needed to evaluate expressions which must be constant, e.g.

```
int a[sizeof(int)*4];
```

In standard C the array dimension must be constant and computable at compile time, but can be expressed by any arbitrary expression, e.g. `int a[2*2]` vs `int a[3+1]`. We would need to perform this computation during semantic analysis, so we can establish the type of the variable a. We can not wait until after the IR is generated. In C99, variably modified array types are allowed for automatic variables (as we have seen, these actually result in imperative code as a result of the declaration), but because of linker restrictions, they are not allowed for static or extern types, and thus the same sort of constant requirement is imposed. So, we still need to be able to determine if the array dimension is a compile-time constant, and this requires constant-folding analysis.

Control Flow Constructs

We have now seen techniques for generating linear 3-address IR for pure expressions (i.e. expressions which do not have implied flow-of-control constructs). We say that these pure expressions are containable in a single **basic block**, which is defined below. Now, we will consider how to generate IR for control flow statements and expressions with hidden control flow such as the && or || operators.

During IR generation, the source-level control constructs such as `if-else` and `for` must be translated into IR form. Some compilers build an AST representation for an entire function or other translation unit, and then descend that tree, generating IR for both control constructs and expressions. Others generate the control flow IR directly during parsing.

The Basic Block

A **Basic Block** is a contiguous sequence of linear IR that has the following properties:
• Each basic block has a unique label (this is not the same as the C language label)
• Therefore the first operation in the basic block can be the target of a branch.
• It ends with a branch (conditional or unconditional), or an exit/function return.
• In some definitions, a function/procedure call is also considered a branch and ends the basic block.
• Other than the first quad, the internal quads of the basic block do not have labels, and therefore it is not possible to branch into the middle of a basic block.
• It must not contain any internal branches, just the branch which terminates the block. Therefore, control flow can not leave in the middle of the block.

The basic block is of vital importance in IR design and optimization. Within a basic block, the optimizer is free to re-arrange the code in any way, so long as it preserves the functionality of the block. It has this freedom because it knows that execution can never possibly enter or leave in the middle of a basic block. Do not confuse "basic block" with a "block" in high-level programming languages such as C.

Representing Basic Blocks

There are many possible ways of storing a linear IR and grouping it into basic blocks. One possible approach is to represent each basic block as a dynamically allocated struct, then have the list of IR codes be attached to that struct either by using a re-allocable array and storing them directly therein, or by creating a linked list. The address of the basic block struct can be used directly in the IR for branch targets.

Most assembly languages use a single-target conditional branch with fall-through if the branch is not taken. In IR design, it is cleaner to make conditional branches have two explicit targets, one for true and one for false. Then in target code generation, one of the branch targets becomes an explicit branch, and the other becomes a fall-through by placing its target basic block immediately after the branch. By storing in each basic block the two possible exiting branch targets, a **Control Flow Graph** is automatically constructed.

When writing a compiler for the C language, it is natural to perform IR generation on a function-by-function basis. After IR generation is completed, much of the semantic information which is limited to the scope of the function may be discarded. It is conceivable that the entire function contains only pure expressions and can therefore be contained in a single basic block. In most cases, however, functions require multiple basic blocks. There is always an initial basic block which represents the start of execution of the function.

After quads have been generated into basic blocks for the entire function, the control flow graph can be searched and the basic blocks ordered to place the ‘‘false’’ targets of basic blocks as their immediate successors, thus reducing it to a single branch target representation.

Another technique is to represent branches explicitly as IR instructions which will necessarily be the last instruction of a basic block. Then, typically, the conditional branches are represented as having just a "TRUE" target, with the "FALSE" target being a fall-through (to the next basic block). This representation is closer to what most processors actually provide, but may obscure opportunities for basic block reordering.

It is typical of C compilers to work one function at a time. This is because C functions could wind up being called by "foreign" code. `main()` is the root of the CFG for the entire program, in most cases, but it is possible to write a C program which starts at a different place! The CFG for a C program therefore is usually not the entire program, but one CFG for each function. Some C compilers attempt to look at inter-function control and data flow. This is a difficult problem.

Lists of statements, declarations and simple branches

Recall that the C language specification states that a function definition consists of a declaration of the function itself (including the return type and arguments), followed by a block (compound statement), which contains a list of statements and declarations (prior to C99, declarations were required to come before the first statement).

Most declarations do not cause the generation of IR code, they merely define the meaning of an identifier to the semantic analysis phase. There are some exceptions. Declarations of variable-length arrays require emission of code to calculate the array size and allocate space for the array.

Some external IRs, such as LLVM, emit declarative code when a declaration is processed. This is because some aspects of a variable’s type need to be passed on into the IR.

Declarations of automatic variables which include an initializer result in the emission of code to calculate the initializer value and assign it to the variable (they are similar to a declaration immediately followed by an assignment). Initialized declarations of global variables do not cause IR to be generated, because the initializer value is required to be known at compile-time. The compiler must emit an assembly-language pseudo-opcode to associate the name of the identifier with its initialized value. We will cover this in another unit.

In generating IR for a compound statement, we simply generate IR for each statement, in order. To do so requires that we maintain a "cursor" of state information. This will include the basic block into which IR code should be emitted (implicit in which is the fact that code is always appended to the end of current basic block), and other state information such as the nearest enclosing switch or loop statement, if any.

A C language label may be attached to any statement (including a compound statement). Recall that labels exist in a namespace separate from other identifiers, and that they are not required to be declared before use. By definition, a labeled statement must cause the termination of the current basic block, and the creation of a new current basic block. In the symbol table, the association between that new block and the label must be recorded. It will turn out that, during final code generation, the assembly language label emitted can not be the actual label used in the program, because of possible conflicts with global variables, function names and other symbols. At that time, an assembly-language label name will need to be generated. E.g. the GNU CC compiler uses internal labels of the form `.L0`, `.L1`, etc. for each basic block. Since these names start with a period character, they can not conflict with C language identifiers.

Note that it is possible for multiple labels to be attached to the same statement:

```
          label1:
          label2:
          label3:
                    a=b;
```

After processing the label, code generation continues for the attached statement in the new basic block. An unconditional branch must be recorded linking the end of the previous basic block to the start of the new one.

A `goto` statement also causes the termination of the basic block, with an unconditional branch to the basic block associated with the label provided in the goto statement. If this label has not yet been seen, an empty basic block can be created and the identity of this block associated with the label in the symbol table. Then, when the target label is later seen, this basic block can be used as the new basic block instead of creating one. After a goto statement, it is still necessary to continue processing subsequent statements, with a new basic block created. This new block is not yet linked to any other block. This does not necessarily mean that it will remain unreachable. Consider (yes, this is an endless loop):

```
f()
{
          /*stmt 1a */
          /*stmt 1b */
          goto label3; /* forward declaration, create incomplete sym tab entry*/
  label2:                    /* create complete symtab entry */
          /*stmt 2a */
          /*stmt 2b */
  label3:          /* complete previously installed symtab entry */
          /*stmt 3a */
          /*stmt 3b */
          goto label2;       /* look up previously installed symtab entry */
}
```

After the IR has been generated for an entire function, we must examine all of the C language statement labels in the symbol table for that function. Labels that were defined but never used by a goto are a warning, but the converse is an error. Likewise, when we traverse the control flow graph (linkages of basic blocks) to create the target code, we will learn if there are any basic blocks which can never be reached. These too should generate a warning.

if-then-else

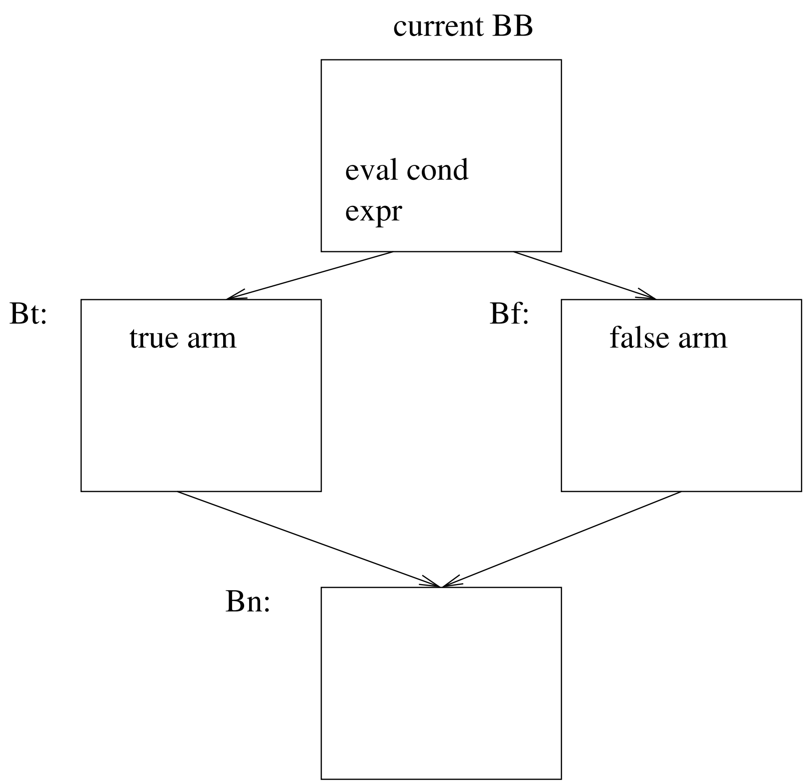
There are three parts to an if-then-else statement (of course, in C there is no `then` keyword): the conditional expression, the true arm, and the false arm (if present). A possible algorithm for generating the IR is as follows: When, in the course of processing statements, we see an IF statement, we create three new basic blocks. Let us called them Bt, Bf and Bn. These will represent the TRUE arm, the FALSE arm, and the statement which follows the `if` statement. If there is no FALSE arm, then Bf and Bn are the same.

We introduce a new routine, call it `gen_condexpr(E,Bt,Bf))`, where `E` is the conditional expression AST. Instead of attempting to produce an rvalue, lvalue or address (as we have seen above for expressions), the goal of `gen_condexpr` is to evaluate the expression and branch to either the true target or the false target (these targets being supplied as parameters). The code for evaluating the conditional expression will be generated in the current basic block (although evaluation may cause additional basic blocks to arise, e.g. if there are && operators). It will create the conditional branch at the end of the basic block to the Bt and Bf targets. We can then recurse on the true and false arms, setting the initial current basic block to Bt or Bf respectively. Each of these arms may create additional basic blocks. Whatever the current basic block is at the conclusion of each arm gets linked to the basic block Bn which will hold the statement following the IF node.

```
gen_if(if_node)
{
          Bt=new_bb();
          Bf=new_bb();
          if (if_node->else_arm)
          {
                    Bn=new_bb();
          }
          else
                    Bn=Bf;
          gen_condexpr(if_node,Bt,Bf); //creates branches to Bt,Bf
          cur_bb=Bt;
          gen_stmt(if_node->then_arm);
          link_bb(cur_bb,ALWAYS,Bn,NULL);
          if (if_node->else_arm)
          {
                    cur_bb=Bf;
```

```
                gen_stmt(if_node->else_arm);
                link_bb(cur_bb,ALWAYS,Bn,NULL);
        }
        cur_bb=Bn;
}
```

Note that with nested IF statements, it is possible that empty basic blocks will get created. These are of no real concern, as they are easily eliminated during subsequent optimization passes and will not bloat the final code.



Evaluation of Comparison Operators in a Conditional Context

`gen_condexpr()` is responsible for generating code to evaluate conditional expressions. If its target is already a boolean expression, such as an == or < operator, the task is fairly simple. The equality and comparison operators translate to a `CMP` quad, which models a similarly named instruction that almost all processors have. The CMP is followed by a branch on the appropriate condition code. E.g. we could designate condition codes `EQ,NE,LT,LE,GT,GE`.

The CMP operation is a subtraction, with the result discarded. Comparisions between signed numbers differ from those between unsigned number. Consider comparing the 16-bit integer 0xFFFF to 0x07FF. If these bit patterns are interpreted as unsigned, then the first one is greater than the second. If, however, they are signed (two's complement), then the first one is -1, which is less than the second one.

Most processors maintain Carry, Zero, Negative and Overflow flags which are set or cleared by arithmetic operations such as CMP, ADD or SUB. The Carry flag is a carry/borrow out of the Most Significant Bit (MSB). The Negative flag is simply the value of the sign bit (MSB) or the result. The Zero flag indicates if the result is all 0 bits. Finally the Overflow (V) bit is set if the operation resulted in a twos complement signed overflow, i.e. the carry/borrow status out of the MSB XOR with the carry/borrow status from the second most significant bit to the MSB. The V bit indicates that the N bit "can't be trusted" because the overflow wrongly flipped the sign bit.

EQ and NE conditions are evaluated by examining the Z flag. For greater/ less than comparisons, the procedure depends on whether the operands are being treated as signed or unsigned. For unsigned comparisions, only the Carry flag is examined, and the first argument is less than the second if the C flag is set. For signed, the first operand is less than the second if (N&&!V || !N&&V), i.e. N XOR V.

Signed Compare

−128<127	127 < −128	−1 < −2	−1 < 1	1 < −1
10000000	01111111	11111111	11111111	00000001
− 01111111	10000000	11111110	00000001	11111111
00000001	11111111	00000001	00000001	00000010
C=0	C=1	C=0	C=0	C=1
V=1	V=1	V=0	V=0	V=0
N=0	N=1	N=0	N=1	N=0
Res: <	Res: >	Res: >	Res: <	Res: >

We can designate additional condition codes, such as `LEU`, to distinguish signed from unsigned comparison. The selection of signed vs unsigned condition codes follows from the analysis of the type of the comparison expression. When unsigned and signed types are compared, the rules can be counter-intuitive. The reader is referred to H&S or the C standard for a more complete discussion.

In these examples which follow, there is an implicit data flow of condition codes from the CMP operation which sets them, to the branch which references them. Another IR philosophy is to represent condition codes explicitly, as if they were another form of temporary value. E.g.

```
%CC100=   CMP       a,10
          BR        %CC100,GE,BB20,BB30
```

If the target of `gen_condexpr()` is a numeric expression, that must be evaluated as an rvalue, and then the result must be compared against 0. E.g. `if (a+b)` is equivalent to `if ((a+b)!=0)`

Condition Inversion

Consider this if-then statement with no else arm:

```
        if (a<b)
                g++;
        x=1;
```

```
BB1:    CMP     a,b
        BRLT    BB2,BB3
BB2:    g=      ADD     g,1
        BR      BB3
BB3:
        x=      MOV     1
```

We have noted that most real assembly languages use single-target branches. Once the basic blocks have been filled in, we would re-traverse the control flow graph, concatenating basic blocks. Whenever there is an unconditional branch, that can be eliminated and the target basic block can simply be placed right after the branch (assuming the target block has not yet already been placed), taking advantage of "fall-through". Likewise, any two-target true/false branches are simplified to be single-target, true-only branches. The false leg simply becomes the next basic block in the linear sequence, again utilizing fall-through.

In the example above, we would be unable to eliminate the `BR bb3`, because the false leg of the BRLT quad is bb3, and therefore bb2 could not follow bb1, but instead would have to be isolated somewhere else in the linear output, resulting in an extra, wasteful branch. But, if the condition code is reversed:

```
BB1:
        CMP     a,b
        BRGE    bb3,bb2
BB2:
g=      ADD     g,1
        BR      bb3
BB3:
x=      MOV     1
        ...
```

Now, bb2 , which is the true leg of the if-then statement, winds up as the false leg of the branch. This doesn't change the meaning of the program. It is effectively a double negation. But it allows bb2 to follow bb1, resulting in the following linearization:

```
bb1:
        CMP     a,b
        BRGE    bb3
bb2:
g=      ADD     g,1
bb3:
        ...
```

Conditional Expressions in an rvalue context

Consider

```
f()
{
        g= (a<b);
}
```

Here the conditional expression `a<b` does not appear in a control flow context. Instead it must generate a value 0 or 1. One way to code this is (note conditional inversion from implied if-then control flow):

```
bb1:
        CMP     a,b
        BRGE    bb3,bb2
bb2:
g=      MOV     1
        BR bb4
bb3:
g=      MOV     0
        BR bb4
bb4:
```

Another choice might be to add a feature to the IR which allows direct access to the condition codes. Below we introduce an operation `CC_LT` which generates a value of 1 if the condition codes indicate LT, and 0 otherwise.

```
bb1:
        CMP     a,b
g=      CC_LT
```

Some processors have an instruction which does this directly, e.g. the `setcc` instructions on the X86 architecture. If such an instruction is lacking, but access to the condition codes register is possible, it could be emulated by masking and shifting the appropriate condition codes flags bits. Branches tend to be expensive on modern processors, so replacing the branch with straight-line code may be a good optimization.

Short-circuit operations

In many programming languages, a construct such as:

```
IF (A<B AND D>E)
```

does not define the order in which the two operands to the logical AND operator are evaluated. On the other hand, C is among those languages which specifically defines the order using the so-called "short-circuit" rule. In the expression `(a<b && d>e)`, the subexpression `a<b` is always evaluated first. If this expression is false, there is no need to evaluate `d<e`, because it doesn't matter anymore. This isn't just laziness, short-circuit rules are a traditional C metaphor for handling a potentially unsafe evaluation:

```
char *p;

        if (p && *p=='X')   {d++;}
```

The expression `*p=='X'` will never get evaluated (and therefore `*p` will not get evaluated) if p is NULL.

In terms of IR generation, logical operators create additional hidden basic blocks. The code above is equivalent to:

```
          if (p)
          {
                    if (*p=='X')

                              d++;
                    }
          }
```

i.e. gen_condexpr() or gen_rvalue() must create additional implicit conditional branches when the && or || operators are part of a conditional or rvalue expression.

We've already seen how to generate quads for if-then statements, so it should come as no surprise to see something like this:

```
bb1:
          CMP       p,0
          BREQ      bb3,bb2
bb2:
%T1=      LOAD      p
          CMP       T1,'X'
          BRNE      bb3,bb4
bb4:
d=        ADD       d,1
          BR        bb3
bb3:      ...
```

Loops

The C language defines three types of loop: while, do-while and for. Below are some schematic IRs for these constructs. Note that any extra basic blocks which conditional expressions, loop bodies, etc. generate are ignored:

```
while (a<b)
{
          BODY
}

bb2:
          CMP       a,b
          BRGE      bb4,bb3
bb3:
          ...IR for BODY ...
          BR        bb2
bb4:
```

Once again, note the use of conditional inversion to put the IR in a form which lends itself to a single branch target architecture. Note that if, in the BODY of the loop, a `break` statement is encountered, that is equivalent to an unconditional branch to bb3. Likewise a `continue` goes to bb1. Here is an alternate implementation:

```
BB1:     ...
         BR       BB3
BB2:
         ... BODY ...
BB3:     CMP      a,b                #continue point
         BRLT     BB2,BB4
BB4:
         ... break point ...
```

This alternative eliminates one branch in the iteration, and would be faster than the first form if the number of iterations is two or greater, which is usually the case. Note that the condition code is NOT inverted in this case, because the false leg is the fall-through, and the true leg is the branch.

Let's see a do-while loop:

```
do
{
         BODY
} while (a<b)


bb1:
         ...code prior to do loop ...
bb2:
         IR for BODY
bb3:
         CMP      a,b
         BRLT     bb2,bb4
bb4:
         ...
```

Although there does not appear to be a reason for introducing basic block bb2, there could be a `continue` statement inside the loop body and bb2 represents the continue point. The condition code is NOT inverted. But consider this do-while loop with short-circuit operators in the conditional expression:

```
do
{
         BODY
} while (a<b && d<e);

bb1:
         ... code prior to do loop ...
bb2:
         IR for BODY
bb3:     /* continue point */
         CMP      a,b
         BRGE     bb4,bb5
bb5:
         CMP      d,e
         BRLT     bb2,bb4
bb4:
```

Here the first test was inverted, but the second wasn't. What gives? We can use the following method: Pass `gen_condexpr()` a flag indicating whether the statement in which the conditional expression appears naturally "wants" a condition code inversion. This flag is honored for a simple comparison, but if a boolean short-circuit operator (&& or ||) is involved, the left side of that operator is recursively evaluated with inversion requested, then the value of the flag is used to determine inversion of the right side. This process can of course be extended to arbitrarily complex expressions. The reader is invited to contemplate `do { BODY }(while (a<b || c==d && e!=f);`

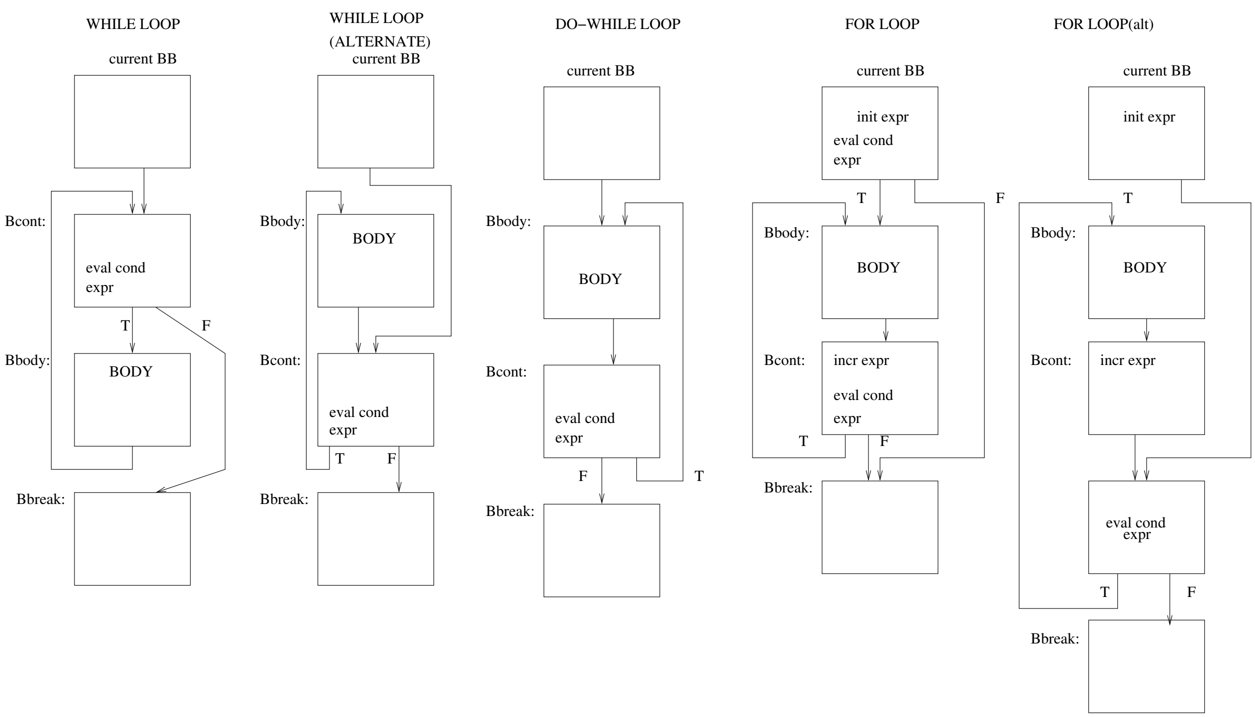
Now let's look at the most complicated loop statement:

```
for(a=1;a<40;a++)
{
        BODY
}

FIRST FORM:
BB1:
a=      MOV     1
        BR      BB4
BB2:    ...BODY...
        BR      BB3
BB3:    #continue point
a=      ADD     a,1
BB4:
        CMP     a,40
        BRLT    BB2,BB5
BB5:    #break point


SECOND FORM:
bb1:
a=      MOV     1
        CMP     a,40
        BRGE    bb4,bb2
bb2:
        IR for BODY
        BR      bb3
bb3:
a=      ADD     a,1
        CMP     a,40
        BRLT    bb2,bb4
bb4:
        ...
```

In both forms, the condition code is not inverted, because the true leg is the branch back. The first form uses fewer quads, but the second form will be slightly faster when the number of iterations is small, because it avoids the cost of the extra branch, at the expense of duplication the "test" part (middle leg) of the for loop. Note that in the second form, a trivial optimization is possible via constant propagation. It is known that `a` contains the value 1 and therefore the outcome of the compare and branch is determinable at compile time, and can be eliminated.



break, continue, return

It should be fairly obvious that the break and continue statements are exactly equivalent to a goto, to the internal basic block label representing the appropriate point in the loop (for the continue) or the statement following the loop (for break). This requires, as part of our state "cursor", keeping track of the current break and continue points of the closest enclosing loop statement. There is an additional complication because of the overloading of the break statement between loops and switch statements (see below).

We already know that a return statement must terminate the basic block. Where return statements exist in the middle of a function, we could explicitly place them in the IR as a RETURN opcode. We had said earlier that every basic block is to be tagged with the possible exit branches, either a single unconditional branch, or a true/false pair. With a RETURN, we could adopt the convention that there are 0 exit branches, designating an implied branch to the EXIT node of the control flow graph for the function. Alternatively, we could designate one basic block as the exit node for the function, and return statements can be implemented as a branch to that exit block.

switch statements

The switch statement in C can be thought of as a series of if tests:

```
switch(c)
{
 case 1:
          /*...*/
          break;
 case 2:
          /*...*/
          break;
 default:
          /*...*/
          break;
}

equivalent:

if (c==1) goto case_1;
else if (c==2) goto case_2;
else goto case_default;
{
 case_1:
          /*...*/
          goto end_switch;
 case_2:
          /*..*/
          goto end_switch;
 case_default:
          /*..*/
          goto end_switch;
}
end_switch:;
```

Although traditional C stylistic guidelines make us write switch statements in the manner shown above, case labels are in fact just another form of label, which only happen to be valid inside of a switch statement. There is no requirement that the statement attached as the body of a switch be a compound statement, nor is there any restriction against branching into that statement:

```
          /*...*/
          goto plain_label;
          /*...*/

          switch(state)
           default: for(i=0;i<k;i++)
                   {
                             /*...*/
                             case STATE_INIT:
                                       /*...*/
                             plain_label:/*...*/
                                       /*...*/
                             case STATE_CLOSING:
                             case STATE_WAITING:
                                       /*...*/
```

```
                }
```

Another strange example:

```
        switch(x)
                default:
                        if (prime(x))
                         case 2:
                         case 3:
                         case 5:
                                process_prime(x);
                        else
                         case 4:
                         case 6:
                                process_comp(x);
```

Such usage is strange and not recommended, but the compiler must understand it.

It is easy to see that this model of generating code for a switch statement will lead to a lot of conditional branches. A traditional way of optimizing this is to use a **computed goto**. Let us employ the non-standard GCC extension of the && operator, which creates a pointer to a statement label. Then the two-case example above might be equivalent to:

```
void *__temp[2]={&&case_1,&&case_2};
        if (c<1 || c>2) goto  case_default;
        else goto *__temp[c-1];
```

If the number of cases is large and the values are contiguous (or nearly so with only a few gaps) this method can be a considerable optimization. It requires that, at code generation time, a table of addresses be constructed. This table is referenced only by an internal label which the compiler creates. (The above example is simplified but it should be fairly self-evident how to extend it for any contiguous range).

If there are many case values but they are sparse, it may be most efficient to place all of the values and branch targets, in sorted order, into a table, and use binary search to select the target.

The compiler must decide when it is more efficient to use a lookup table vs a sequence of tests. This decision may be difficult to make, because at the time of IR generation, knowledge of the relative expense of the various assembly-language operations may not be known. Furthermore, using a lookup table requires the compiler to extend its notion of basic block exits to include an arbitrary number of possible targets rather than just two as heretofore illustrated, and to include some method in the IR of encoding the jump table.

switch statements add to our code generation state "cursor". Within the body of the switch statement, the break target must be set to the basic block which will follow the switch. The body of the switch statement should be generated first, in a new basic block. During the course of that generation, any statements with a case or default label attached should be noted and a list of such labels associated with the enclosing switch statement (in any other context, case and default labels are not valid). Each of these labels will, of course, introduce a new basic block. The compiler must keep track of which label value is associated with which block. It must also detect errors such as duplicate values. If no default label is found within the switch statement body, one must be created which points to the next basic block. Then the code to evaluate the switch expression and perform the branching (whichever algorithm is selected) should be generated into the original basic block. (this may cause that basic block to split one or more times as a result of the branches)

Function Calls

Functions are very self-contained in C, unlike in many other languages where they can be nested within each other. Functions are the basic unit of inter-file linkage. The compiler can not generally assume any knowledge of the function being called, other than what is known from its prototype.

Although control always returns to the point after the function call, it would be appropriate to terminate the current basic block and begin a new one. This will simplify some data flow analysis assumptions, since once control leaves the current basic block to enter another function, it is possible that global variables may be modified.

We'll have to defer a lot of discussion of function calls to a later unit. The function call sequence consists of 7 parts: the caller saving whatever registers need to be preserved, the caller evaluating and placing the arguments in their proper places, the callee saving registers, the callee placing the return value in the correct place, the callee restoring registers which it saved, the caller extracting the return value, and the caller restoring registers which it saved. At the IR level, we are not concerned with registers, so let's look at the other phases.

Clearly a function call can not be represented by a single traditional quad, because it has a potentially large number of arguments. Code is generated in the caller to evaluate each argument. The C standard says that the order of argument evaluation is not specified. This is a common source of programmer error if side-effects exist in the arguments:

```
main()
{
 char *p="ABC";
         printf("%c %c %c\n",*p++,*p++,*p++);
}
```

On the X86-32 architecture, arguments are passed on the stack, with the left-most argument being pushed *last*. This ensures that the left-most argument will be closest to the top of the stack as viewed by the callee. In the event that more arguments are passed than the callee expects, no ill effects result. If, however, the left-most argument were pushed first, then the callee would not be able to know its position with respect to the stack pointer on entry to the function if the function takes a variable number of arguments.

Most compilers for X86-32 would therefore prefer to evaluate the arguments in the same right-to-left order in which they will be pushed. This avoids excessive temporary variables. The code above therefore prints out the counter-intuitive "CBA".

On the other hand, in the SPARC architecture, arguments are generally passed in registers. The left-most argument goes in register %o0, the next one in %o1, etc. Compilers for this architecture will probably want to evaluate left-to-right, and the code above outputs "ABC".

Given the architecture dependence of function argument passing, we'd want to introduce a fairly flexible and neutral way of representing it in IR. We might introduce the IR instruction ARGBEGIN, which has no destination and one source operand giving the total number of arguments which follow. Then an ARG instruction with a single source operand which is used to "push" the argument, and finally a CALL instruction with the address of the function to call. The order of the arguments could be left implicit (or an explicit argument number could be squeezed in to the ARG instruction.

```
/* example of a function call */
         z(a,b)

BB100:
         ARGBEGIN  2
         ARG       1,a
         ARG       2,b
         CALL      z
         BR        BB101             /* Because the fncall terminates the BB */
         /* This fall-through would be optimized out later and the
                   above BR quad would not generate an equivalent asm opcode */
BB101:
```

On the callee side, formal parameters are just another form of variable. We don't need to do anything different from an IR standpoint to access these parameters; they are essentially another "address mode," and analogously we have chosen not to worry about local vs global variable addressing at this time.

We need a RETURN instruction in the IR which accepts a single source operand. The issuance of a RETURN terminates the current basic block. A basic block thus terminated should be noted as it forms an EXIT node for the control flow graph of the function. An alternative notation is that the RETURN quad sets the return value from the function, and is followed by an explicit branch to the exit node of the function. The optimizer or back end could later recognize this and emit a `ret` instruction which accomplishes both things. This latter approach allows for there to be a single point where control exits a function.

The original C definition did not allow structures to be returned from functions. Based on our previous discussion, we can see the problem: structures are not scalar types, and so returning a structure breaks our RETURN model. We'll have to discuss this later when we look at specific architectures and calling conventions.