CS143 Handout 22
Summer 2010 July 23rd, 2010

Intermediate Representation

Handout written by Maggie Johnson and revised by Julie Zelenski.

Most compilers translate the source program first to some form of *intermediate representation* and convert from there into machine code. The intermediate representation is a machine- and language-independent version of the original source code. Although converting the code twice introduces another step, use of an intermediate representation provides advantages in increased abstraction, cleaner separation between the front and back ends, and adds possibilities for re-targeting/cross-compilation. Intermediate representations also lend themselves to supporting advanced compiler optimizations and most optimization is done on this form of the code.

There are many intermediate representations in use (some suggests it may be as many as a unique one for each existing compiler) but the various representations are actually more alike than they are different. Once you become familiar with one, it's not hard to learn others. Intermediate representations are usually categorized according to where they fall between a high-level language and machine code. IRs that are close to a high-level language are called high-level IRs, and IRs that are close to assembly are called low-level IRs. For example, a high-level IR might preserve things like array subscripts or field accesses whereas a low-level IR converts those into explicit addresses and offsets. For example, consider the following three code examples (from Muchnick), offering three translations of a 2-dimensional array access:

Original	*High IR*	*Mid IR*	*Low IR*
`float a[10][20];`	`t1 = a[i, j+2]`	`t1 = j + 2`	`r1 = [fp - 4]`
`a[i][j+2];`		`t2 = i * 20`	`r2 = [r1 + 2]`
		`t3 = t1 + t2`	`r3 = [fp - 8]`
		`t4 = 4 * t3`	`r4 = r3 * 20`
		`t5 = addr a`	`r5 = r4 + r2`
		`t6 = t5 + t4`	`r6 = 4 * r5`
		`t7 = *t6`	`r7 = fp - 216`
			`f1 = [r7 + r6]`

The thing to observe here isn't so much the details of how this is done (we will get to that later), as the fact that the low-level IR has different information than the high-level IR. What information does a high-level IR have that a low-level one does not? What information does a low-level IR have that a high-level one does not? What kind of optimization might be possible in one form that might not in another?

High-level IRs usually preserve information such as loop-structure and if-then-else statements. They tend to reflect the source language they are compiling more than lower-level IRs. Medium-level IRs often attempt to be independent of both the source language and the target machine. Low-level IRs tend to reflect the target architecture very closely, and as such are often machine-dependent. They differ from actual assembly code in that there may be choices for generating a certain sequence of operations, and the IR stores this data in such a way as to make it clear that choice must be made. Sometimes a compiler will start-out with a high-level IR, perform some optimizations, translate the result to a lower-level IR and optimize again, then translate to a still lower IR, and repeat the process until final code generation.

Abstract Syntax Trees

A parse tree is an example of a very high-level intermediate representation. You can usually completely reconstruct the actual source code from a parse tree since it contains all the information about the parsed program. (It's fairly unusual that you can work backwards in that way from most IRs since much information has been removed in translation).

More likely, a tree representation used as an IR is not quite the literal parse tree (intermediate nodes may be collapsed, groupings units can be dispensed with, etc.), but it is winnowed down to the structure sufficient to drive the semantic processing and code generation. Such a tree is usually referred to as an *abstract syntax tree*. (The opposite, a tree that provides a 1:1 mapping without collapsed nodes and with all terminals at the leaves is called a *concrete syntax tree*.) In the programming projects so far, you have already been immersed in creating and manipulating such an abstract syntax tree. Each node represents a piece of the program structure and the node will have references to its children subtrees (or none if the node is a leaf) and possibly also have a reference to its parent.

Consider the following excerpt of a programming language grammar:

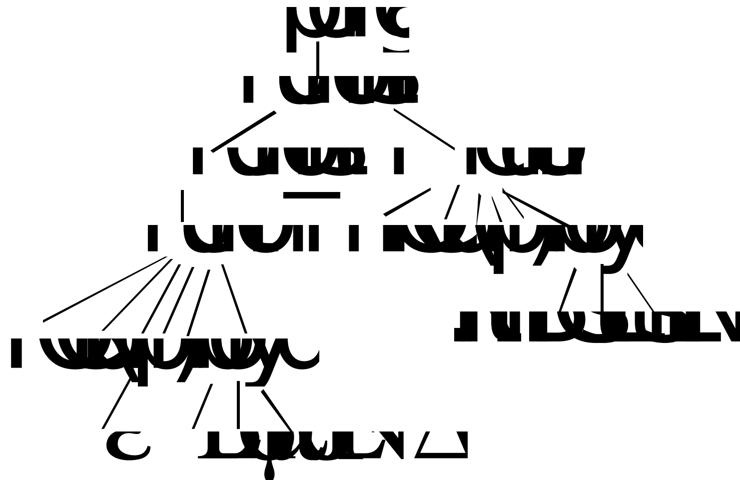
```
program         ->   function_list
function_list   ->   function_list function | function
function        ->   PROCEDURE ident ( params ) body
params          ->   ...
```

A sample program for this language:

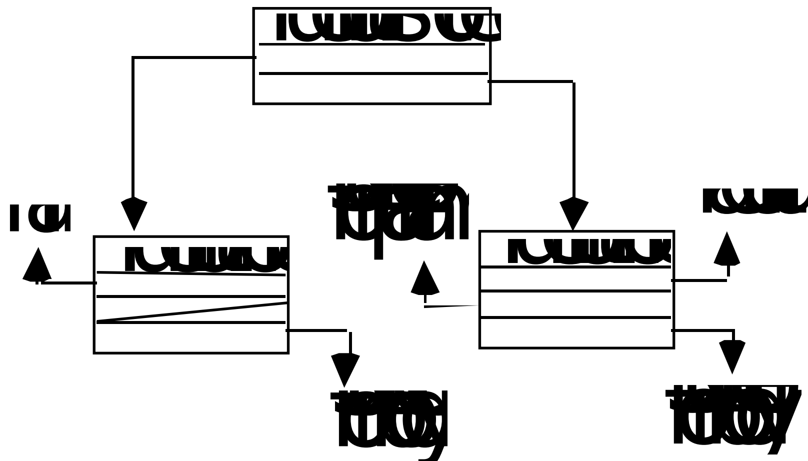
```
PROCEDURE main()
BEGIN
   statement...
END
```

```
PROCEDURE factorial(n:INTEGER)
BEGIN
   statement...
END
```

The literal parse tree (or concrete syntax tree) for the sample program looks something like:



Here is what the abstract syntax tree looks like (notice how some pieces like the parens and keywords are no longer needed in this representation):



The parser actions to construct the tree might look something like this:

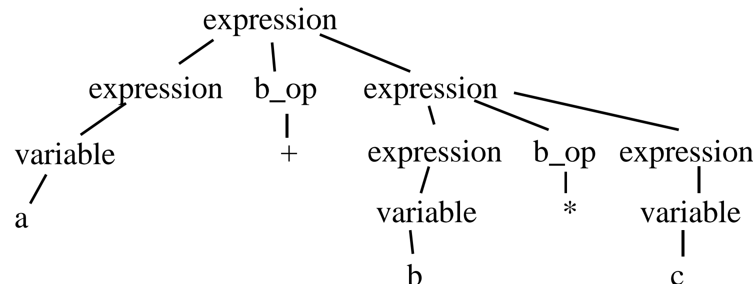
```
function: PROCEDURE ident ( params ) body
            { $$ = MakeFunctionNode($2, $4, $6); }

function_list: function_list function
            { $$ = $1; $1->AppendNode($2); }
```

What about the terminals at the leaves? Those nodes have no children; usually these will be nodes that represent constants and simple variables. When we recognize those parts of the grammar that represent leaf nodes, we store the data immediately in that node and pass it upwards so it can participate in the larger tree.

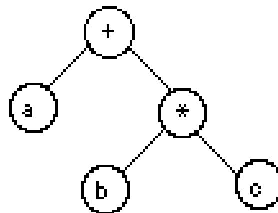
```
constant : int_constant
            { $$ = MakeIntConstantNode($1); }
```

AST To Assembly

With an abstract syntax tree in place, we can now explore how it can be used to drive a translation. As an example application, consider how a syntax tree for an arithmetic expression might be used to directly generate assembly language code. Let's assume the assembly we are working with is for a simple machine that has a collection of numbered registers. Its limited set of operations include **LOAD** and **STORE** to read and write from memory and two-address forms of **ADD**, **MULT**, etc. which overwrite the first operand with the result. We want to translate expressions into the proper sequence of assembly instructions during a syntax-directed translation. Here is a parse tree representing an arithmetic expression:



In going from the parse tree to the abstract syntax tree, we get rid of the unnecessary non-terminals, and leave just the core nodes that need to be there for code generation:



Here is one possible data structure for an abstract expression tree:

```
typedef struct _tnode {
   char label;
   struct _tnode *lchild, *rchild;
} tnode, *tree;
```

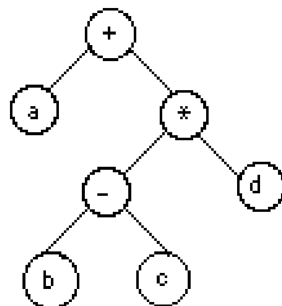
To generate code for the entire tree, we first generate code for each of the subtrees, storing the result in some agreed-upon location (usually a register), and then combine those results. The function **GenerateCode** below takes two arguments: the subtree for

which it is to generate assembly code and the number of the register in which the computed result will be stored.

```
void GenerateCode(tree t, int resultRegNum) {
   if (IsArithmeticOp(t->label)) {
      GenerateCode(t->left, resultRegNum);
      GenerateCode(t->right, resultRegNum + 1);
      GenerateArithmeticOp(t->label, resultRegNum, resultRegNum + 1);
   } else {
      GenerateLoad(t->label, resultRegNum);
   }
}

bool IsArithmeticOp(char ch) {
   return ((ch == '+') || (ch == '-') || (ch == '*') || (ch == '/'));
}

void GenerateArithmeticOp(char op, int reg1, int reg2) {
   char *opCode;
   switch (op) {
      case '+': opCode = "ADD";
              break;
      case '-': opCode = "SUB";
              break;
      case '*': opCode = "MUL";
              break;
      case '/': opCode = "DIV";
              break;
   }
   printf("%s R%d, R%d\n", opCode, reg1, reg2);
}

void GenerateLoad(char c, int reg) {
   printf("LOAD %c, R%d\n",c , reg);
}
```

In the first line of **GenerateCode**, we test if the label of the root node is an operator. If it's not, we emit a load instruction to fetch the current value of the variable and store it in the result register. If the label is an operator, we call **GenerateCode** recursively for the left and right expression subtrees, storing the results in the result register and the next higher numbered register, and then emit the instruction applying the operator to the two results. Note that the code as written above will only work if the number of available registers is greater than the height of the expression tree. (We could certainly be smarter about re-using them as we move through the tree, but the code above is just to give you the general idea of how we go about generating the assembly instructions).

Let's trace a call to **GenerateCode** for the following tree:



The initial call to **GenerateCode** is with a pointer to the '+' and result register 0.

```
GenerateCode('+', 0)
   GenerateCode('a', 0)
      write "LOAD a, R0"
   GenerateCode('*', 1)
      GenerateCode('-', 1)
         GenerateCode('b', 1)
            write "LOAD b, R1"
         GenerateCode('c', 2)
            write "LOAD c, R2"
         write "SUB R1, R2"
      GenerateCode('d', 2)
         write "LOAD d, R2"
      write "MUL R1, R2"
   write "ADD R0, R1"
```

We end up with this set of generated instructions:

```
LOAD a, R0
LOAD b, R1
LOAD c, R2
SUB R1, R2
LOAD d, R2
MULT R1, R2
ADD R0, R1
```

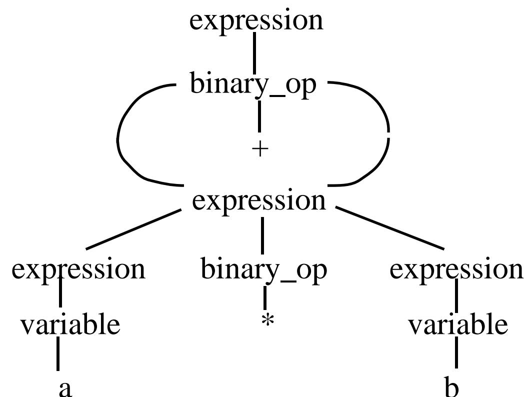
Notice how using the tree height for the register number (adding one as we go down the side) allows our use of registers to not conflict. It also reuses registers (R2 is used for both c and d). It is clearly not the most optimal strategy for assigning registers, but that's a topic for later.

Direct Acyclic Graphs

In a tree, there is only one path from a root to each leaf of a tree. In compiler terms, this means there is only one route from the start symbol to each terminal. When using trees as intermediate representations, it is often the case that some subtrees are duplicated. A logical optimization is to share the common subtree. We now have a data structure with more than one path from start symbol to terminals. Such a structure is called a

directed acyclic graph (DAG). They are harder to construct internally, but provide an obvious savings in space. They also highlight equivalent sections of code and that will be useful later when we study optimization techniques, such as only computing the needed result once and saving it, rather than re-generating it several times.

```
a * b + a * b;
```



gcc's Intermediate Representation

The **gcc/g++** compiler uses an abstract syntax tree to capture the results from its **yacc/bison** parser. The compiler is written in C, not C++, thus the nodes are not C++ objects, like Decaf nodes. Instead, each tree is a C pointer to a structure with a tag field to identify the node type (function, for loop, etc,) and there are various functions and macros to pick out the individual fields that are appropriate for each type of node. The tree representation is used for each function. Once a function is parsed the tree representation of that function is translated to an intermediate language called RTL (register-transfer language). RTL is inspired by Lisp lists. It has both an internal form, made up of structures that point to other structures, and a textual form that is used in the machine description and in printed debugging dumps. The textual form uses nested parentheses to indicate the pointers in the internal form. Here is the RTL output for a simple hello, world program:

```
;; Function main

(note 3 2 4 "" NOTE_INSN_FUNCTION_BEG)
(note 6 4 7 0 NOTE_INSN_BLOCK_BEG)
(insn 7 6 8 (set (reg:SI 106)
        (high:SI (symbol_ref:SI ("*.LLC0")))) -1 (nil)
    (nil))
(insn 8 7 10 (set (reg:SI 8 %o0)
        (lo_sum:SI (reg:SI 106)
            (symbol_ref:SI ("*.LLC0")))) -1 (nil)
    (nil))

(call_insn 10 8 12 (parallel[
            (set (reg:SI 8 %o0)
```

```
                (call (mem:SI (symbol_ref:SI ("printf")) 0)
                    (const_int 0 [0x0])))
            (clobber (reg:SI 15 %o7))
        ] ) -1 (nil)
    (nil)
    (expr_list (use (reg:SI 8 %o0))
        (nil)))
(note 12 10 13 0 NOTE_INSN_BLOCK_END)
(note 13 12 15 "" NOTE_INSN_FUNCTION_END)
```

RTL is a fairly low-level IR. It assumes a general purpose register machine and incorporates some notion of register allocation and instruction scheduling. The **`gcc`** compiler does most of its optimizations on the RTL representation, saving only machine-dependent tweaks to be done as part of final code generation.

Java Byte Code

Java's standard compiler (**`javac`**) compiles to *bytecode* which is itself an intermediate representation. Here is the Java bytecode generated for a simple hello, world program.

```
Method Main()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

Method void main(java.lang.String[])
   0 getstatic #2 <Field java.io.PrintStream out>
   3 ldc #3 <String "Hello world">
   5 invokevirtual #4 <Method void println(java.lang.String)>
   8 return
```

Java bytecode is a fairly high-level IR. It is based on stack-based machine architecture and includes abstract notions such as **`getstatic`** and **`invokevirtual`** along with more low-level instructions such as **`ldc`** (load constant) and **`add`**.

Bytecode is targeted to a virtual machine, although there are, in fact, hardware-embodiments of this machine (e.g., the "Java chip" CPU), in most cases, the bytecodes need to be further translated to machine code. The javac compiler does not handle the usual back-end task of converting the IR to machine code, it is instead done by an interpreter within the Java virtual machine (or possibly by a JIT code generator instead of an interpreter). Some compilers translate either Java source code or Java bytecode all the way down to native code (**`gcj`**, for instance).

Bibliography

A. Aho, J.D. Ullman, Foundations of Computer Science, New York: W.H. Freeman, 1992.

A. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Reading, MA: Addison-Wesley, 1986.

J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

S. Muchnick, Advanced Compiler Design and Implementation. San Francisco, CA: Morgan Kaufmann, 1997.

A. Pyster, Compiler Design and Construction. New York, NY: Van Nostrand Reinhold, 1988.