CS143 Handout #25
Winter 2001

Intermediate representations

Handout written by Maggie Johnson and revised by me.

Most compilers translate the source 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 and thus incurs loss in compiler efficiency, use of an intermediate representation provides advantages in increased abstraction, cleaner separation between the front and back ends, adds possibilities for re-targeting/cross-compilation, and works well with many advanced optimization techniques.

There are many intermediate representations in use (one author suggests it may 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 like C, 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 do you see that a high-level IR has 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. TAC, the IR we will be using, is a medium to low-level IR. 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. Often 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

Recall that a parse tree is another way of representing a derivation. You can think of a parse tree as an example of a high-level intermediate representation. In fact, it is often possible to reconstruct the actual source code from a parse tree and the corresponding symbol table. (It's fairly unusual that you can work backwards in that way from most IRs since much information has been removed in translation). If we were to build a tree during the parsing phase, it could form the basis of a syntax tree representation of the input program. Typically, this is not quite the literal parse tree recognized by the parser (intermediate nodes may be collapsed, groupings units can be dispensed with, etc.), but it is winnowed down to the sufficient structure to drive the semantic processing and code generation. Such a tree is usually referred to as an *abstract syntax tree*. Here is an example of a data structure that one might use for such a tree:

```
struct node {
   int nodetype;
   // these fields hold the various subtrees beneath this node
   struct node *field[5];
};
```

Now consider the following excerpt of a grammar for some programming language:

```
program          ←   function_list
function_list    ←   function_list function | function
function         ←   FUNC variable ( parameter_list ) statement
```

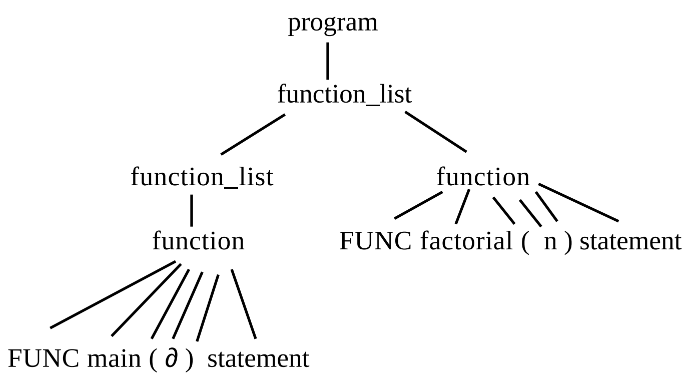
A simple program in this language:

```
FUNC main() {
   statement...
}

FUNC factorial(n) {
   statement...
}
```

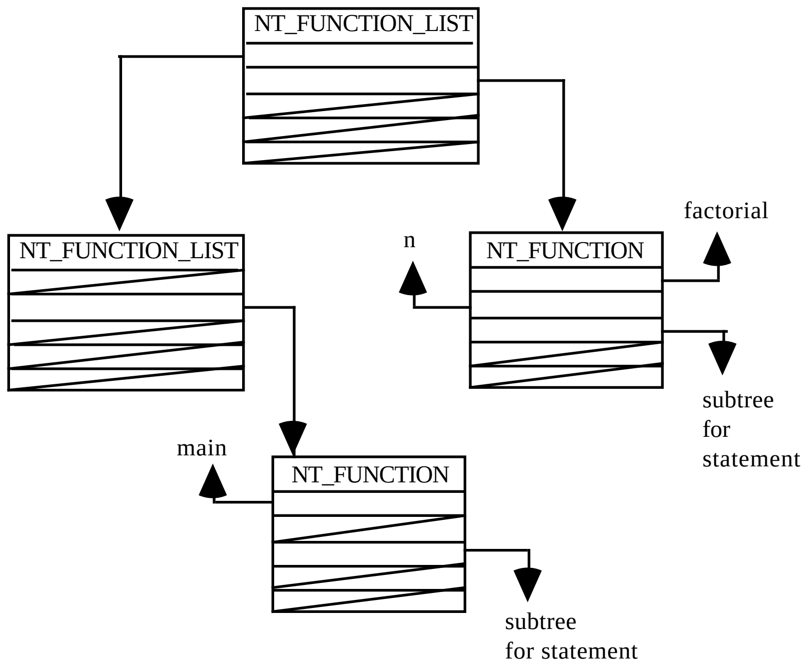
We define node-type constants for all our non-terminals:

```
#define NT_PROGRAM         1
#define NT_FUNCTION_LIST   2
#define NT_FUNCTION        3
...
```

The following parse tree shows a parse of the sample program above:



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



To create a new node in a tree, we could write a function like this:

```
struct node *MakeNode(int type, struct node *f1, struct node *f2,
                      struct node *f3, struct node *f4, struct node *f5)
{
   struct node *t = (struct node *)malloc(sizeof(struct node));

   t->nodetype = type;
   t->field[0] = f1;
   t->field[1] = f2;
   t->field[2] = f3;
   t->field[3] = f4;
   t->field[4] = f5;
   return t;
}
```

The actions associated with the productions would then look something like this:

```
function_list : function_list function
                { $$.tree = MakeNode(NT_FUNCTION_LIST,
                                    $1.tree, $2.tree, NULL, NULL, NULL);}
```

What about the terminals at the leaves? We add fields to our node structure for those that have no children, usually these will be nodes that represent constants and simple variables:

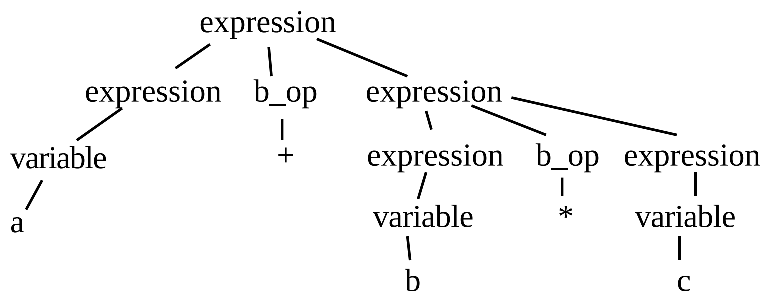
```
struct node {
   int nodetype;
   union {
      int value;        // integer val for int constant leaf node
      char *name;       // name for variable leaf node
      struct node *ptr; // subtree for non-leaf node
   } field1;
   struct node *fields[4]; // other subtrees
};
```

If nodetype is `NT_INTEGER` then `tree->field1.value` holds the value of that integer; if it is a variable, the name is recorded and so on. Here is an idea of what creating a leaf node might look like:

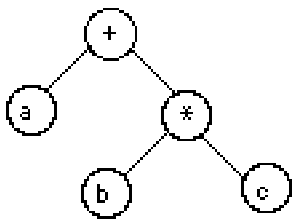
```
constant : int_consant
                { $$.tree = MakeNode(NT_INT_CONSTANT, $1.intVal,
                                     NULL, NULL, NULL, NULL);}
```

Generating assembly code from syntax trees

Now that we have some idea how to represent and build an abstract syntax tree, we can 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 a simple machine a set of numbered registers and a limited set of operations including 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 (now unnecessary) non-terminals, and leave just the core nodes that need to be there for code generation:



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

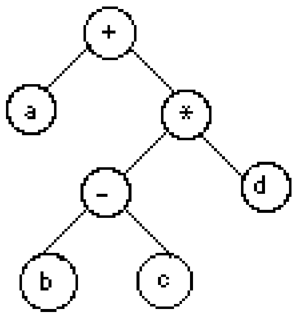
```
typedef struct _tnode {
   char label;
   struct _tnode *lchild, *rchild;
} tnode, *tree;
```

In order to generate code for the entire tree, we will have to 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: a pointer to the root of the subtree for which it must generate assembly code and the number of the register in which the value of this subtree should be computed.

```
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 i, int j)
{
   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, i, j);
}

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

In the first line of `GenerateCode`, we test if the label of the root of the tree 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:

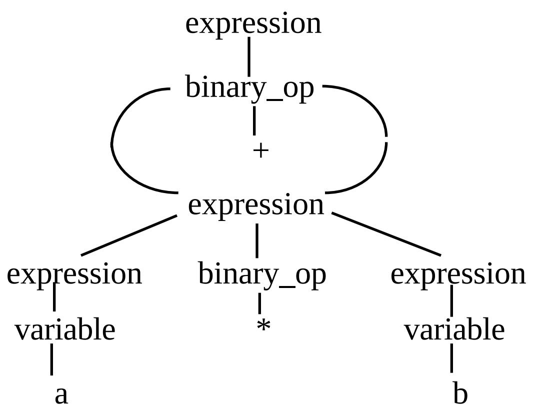
```
LOAD a, R0
LOAD b, R1
LOAD c, R2
SUB R1, R2
LOAD d, R2
MULT R1, R2
ADD R0, R1
```

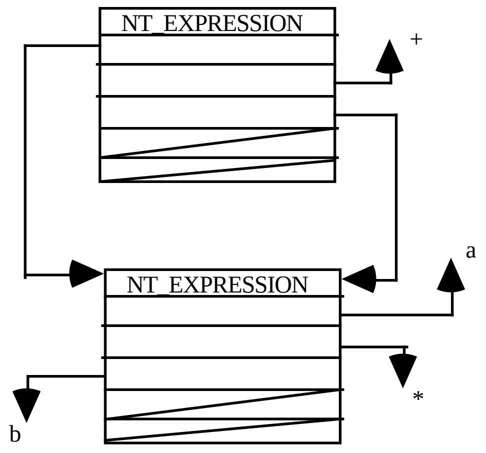
Notice how using of the tree height for the register number (adding one as we go down the side) allows our use of registers to not conflict. It also reuses registers (R2 is used for both c and d), but it is clearly not the most optimal strategy for assigning registers, but that’s a topic for later.

A quick aside: directed 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 sub-tree. 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 bits of code that will be useful later when we study optimization techniques.

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





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.