CS143 Handout #26
Autumn 2000

Intermediate representation and TAC

Handout written by Maggie Johnson and revised by me.

Many modern compilers translate the source first to an *intermediate representation* and 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, using 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 be a unique one for each compiler) but the various representations are actually more alike than they are different, and 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. Typically, a high-level IR preserves things like array subscripts and a low-level IR has 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[20][10];      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 also 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-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 their 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 the appropriate 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). 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
```

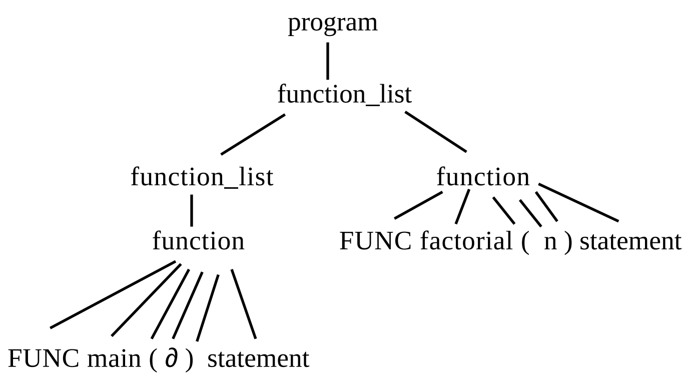
A simple program in this language:

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

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

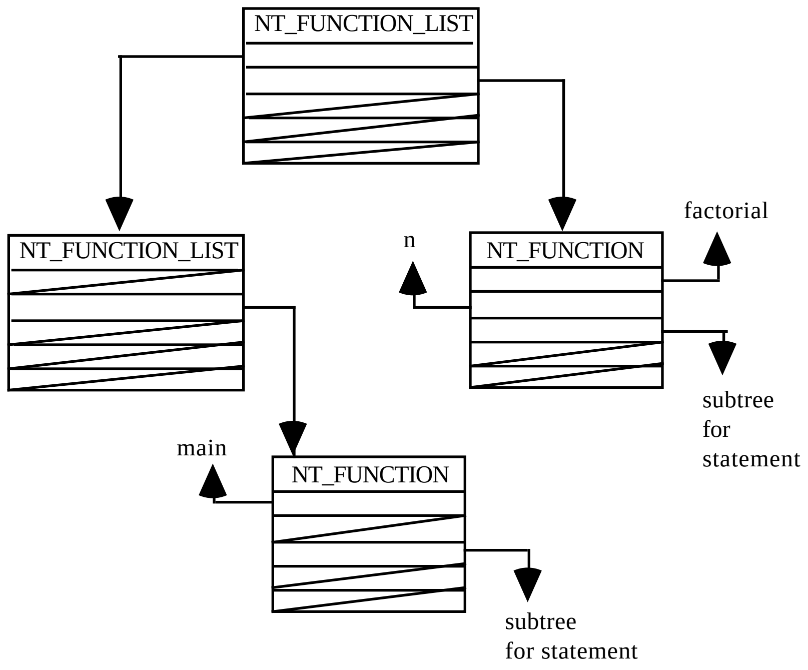
We define node-type constants for all our non-terminals:

```
#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 like this:

```
function_list1 -> function_list2 function
                { function_list1.tree = MakeNode(NT_FUNCTION_LIST,
                     function_list2.tree, function.tree, NULL, NULL, NULL);}
```

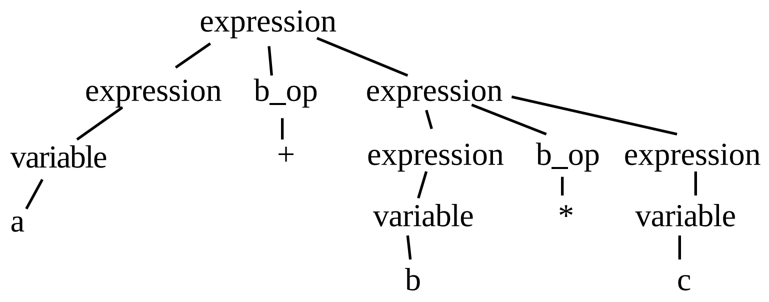
What about the terminals at the leaves? We add fields for to our node structure for those that have no children, usually these will be nodes that represent constants and simple variables:

```
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
};
```

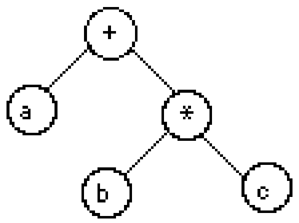
If nodetype is `NT_INTEGER` then `tree->field1.value` holds the value of that integer; if it is a variable, the name is recorded and so on.

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. 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:

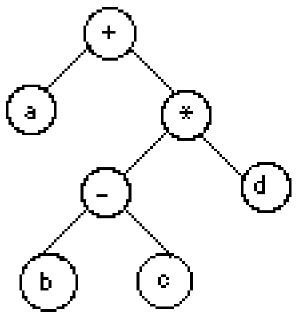
```
typedef struct _tnode {
   char label;
   struct _tnode *lchild, *rchild;
} tnode, *tree;
```

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:

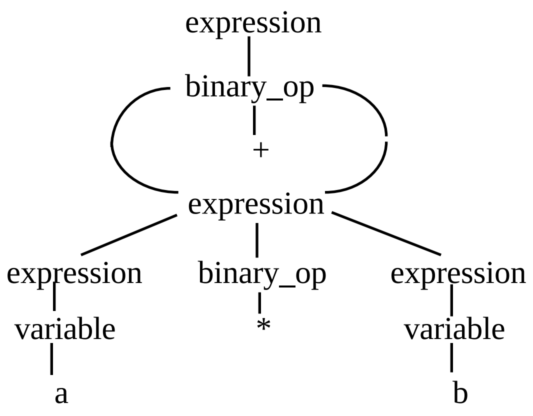
```
LOAD a, R0
LOAD b, R1
LOAD c, R2
SUB R1, R2
LOAD d, R2
MULT R1, R2
ADD R0, R1
```

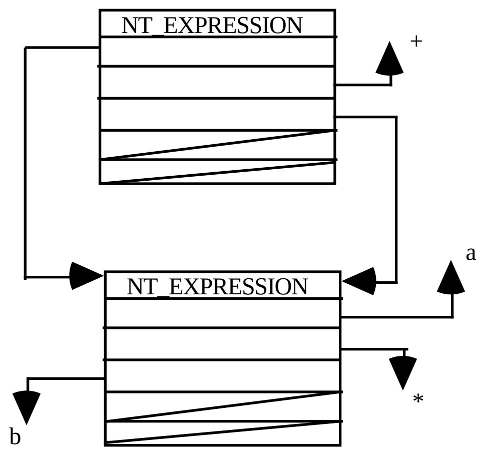
Notice how using of the tree height for the register number (adding one as we go down the side) allows our use of registers to not conflict . It also reuses registers (R2 is used for both c and d), but it is clearly not the most optimal strategy for assigning registers, but that's a topic for later.

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;
```





Three address code

Three-address code (TAC) will be our intermediate representation used in our SOOP compiler. It is basically a sort of generic assembly language that falls in the lower-end of the mid-level IRs. Many compilers use an IR similar to TAC. It is a sequence of instructions, each of which can have at most three operands. The three operands could be two operands to a binary arithmetic op and the third the result location, or two operands to a comparison and a third location to branch to, and so on. For example, here is a C arithmetic expression and its corresponding TAC translation:

```
a = b * c + b * d

t1 = b * c;
t2 = b * d;
t3 = t1 + t2;
a = t3;
```

Notice the use of temp variables that are created by the compiler as needed to keep the number of operands down to three. Of course, it's a little more complicated than the above example, because we have to translate branching and looping instructions, as well as subprogram calls. Here is an example of the TAC for a branching translation:

```
     if (a <= b + 1)
        a = a - c;
     c = b * c;

     t1 = b + 1;
     if a > t1 goto L0;
     t2 = a - c;
     a = t2;
L0:  t3 = b * c;
     c = t3;
```

Notice that the if statement has three addresses too: a, T1 and L0.

SOOP TAC instruction formats

The convention followed in the examples below is that t1, t2, and so on refer to variables and L1, L2, etc. are used for labels. Labels are attached to the instruction that serve as targets for goto/branch instructions, and are used to identify function/method definitions and string constants.

Variable declarations:

```
Var t1 <size>;
```

(if size not specified, assume 4 bytes)

Assignment operations:

```
t2 = t1;
t1 = <integer>;
```

Arithmetic operations:

```
t3 = t2 + t1;
t3 = t2 - t1;
t3 = t2 * t1;
t3 = t2 / t1;
t3 = t2 % t1;
t2 = -t1;
```

Conditional/logical operations:

```
t3 = t2 == t1;
t3 = t2 < t1;
t3 = t2 && t1;
t3 = t2 || t1;
t2 = !t1;
```

(can convert all original source tests to one of the above)

Labels and branches:

```
L1:
Goto L1;
IfZ t1 Goto L1;
ifNZ t1 Goto L1;
```

(branch only has 2 operands, branch test is zero/non-zero)

Function calls:

```
Arg(t1);
Arg(L1);
```

(set arg for function call)

```
L1();
```

(call function with label L1)

```
t1 = L1();
```

(call function and store return value)

Function definitions:

```
BegunFunc <size>;
```

(size is number of bytes in args)

```
EndFunc;
Return t1;
Return;
```

Memory references

```
t1 = *t2;
*t1 = t1;
```

Array indexing:

To access arr[5], add offset multiplied by elem size to base and deref

Object extensions:

```
t1 = This;
```

(t1 holds pointer to this object)

```
t1.<integer>()
```

(call method at index <integer> of vtable)

```
t2 = t1.<integer>()
```

To access ivars, add offset to base, deref

Data specification:

```
String L1 "abcdefg";
VTable L1 = L2, L3, ...;
```

Here is an example of a simple SOOP program and its TAC:

```
void main(void) {
   int a;
   int b;
   int c;
   a = 1;
   b = 2;
   c = a + b;
   Print(c);
}
```

```
main:
        BeginFunc 0     ;
        Var a   ;
        Var b   ;
        Var c   ;
        Var _t0 ;
        _t0 = 1 ;
        a = _t0 ;
        Var _t1 ;
        _t1 = 2 ;
        b = _t1 ;
        Var _t2 ;
        _t2 = a + b     ;
        c = _t2 ;
        // Print
        Arg c   ;
        PrintInteger()  ;
        EndFunc ;
```

As you can see, what we have to do is figure out how to get from one to the other as we parse. This includes not only generating the TAC, but figuring out the use of temp variables, creating labels, calling functions, etc. Since we have a lot to do, we will make the mechanics of generating the TAC as easy as possible. In our parser, we will just write out the TAC instruction one after another to either stdout or a file. We won't bother to create a data structure to hold all the instructions. In addition, we will simplify SOOP a little by excluding doubles and internally treating bools as integers. Classes, arrays, and strings will be implemented with pointers (again, just internally). This means we only ever need to deal with 32-bit variables.

Thus, SOOP has a very simple intermediate-code generation model, although modern compilers simply use more complex variants of ours. As each production is reduced, we will emit the necessary instructions. This makes our code-generation somewhat limited—particularly for the way we would have to do switch statements—but here is the interesting thing. We can translate more-or-less any language structure into an executable program in a single pass, and we never need to go back and edit anything we've output.

To see how an SDT can generate TAC, we need to look at the derivation, and figure out where the different TAC statements should be generated as the productions are reduced. Let's start with a really simple program:

```
void main(void) {
  Print("hello world");
}
```

Here is the TAC:

```
main:
        BeginFunc 0     ;
        String _L0 = "hello world"      ;
        // Print
        Arg _L0 ;
        PrintString()   ;
        EndFunc ;
```

Notice that the string is assigned a label. Also, we call a *library* function called PrintString() to do the actual printing. Here is the derivation of the source program. Identify where and what processing occurs as these productions are reduced to generate the given TAC:

```
DeclarationList ->
Type -> Void
ParameterListOrVoid -> Void
OptVariableDeclarationList ->
Constant -> StringConstant
Expression -> Constant
ExpressionList -> Expression
PrintStatement -> Print ( ExpressionList )
Statement -> PrintStatement ;
StatementList -> Statement
OptStatementList -> StatementList
CompoundStatement -> { OptVariableDeclarationList OptStatementList }
FunctionDefinition -> Type Identifier ( ParameterListOrVoid )
CompoundStatement
Declaration -> FunctionDefinition
DeclarationList -> DeclarationList Declaration
Program -> DeclarationList
```

Here is another simple program with an expression:

```
void main(void) {
   int a;
   a = 2 + a;
   Print(a);
}
```

Here is the resulting TAC:

```
main:
        BeginFunc 0     ;
        Var a   ;
        Var _t0 ;
        _t0 = 2 ;
        Var _t1 ;
        _t1 = _t0 + a   ;
        a = _t1 ;
        // Print
        Arg a   ;
        PrintInteger()  ;
        EndFunc ;
```

Here is the derivation. Again, consider where the instructions above must be emitted relative to the parsing activity:

```
DeclarationList ->
Type -> Void
ParameterListOrVoid -> Void
Type -> Int
ArrayModifiers ->
Variable -> Type Identifier ArrayModifiers
VariableDeclaration -> Variable
VariableDeclarationList -> VariableDeclaration
OptVariableDeclarationList -> VariableDeclarationList
Var -> Identifier
Constant -> IntConstant
Expression -> Constant
Var -> Identifier
```

```
Expression -> Var
Expression -> Expression + Expression
SimpleStatement -> Var = Expression
Statement -> SimpleStatement ;
StatementList -> Statement
Var -> Identifier
Expression -> Var
ExpressionList -> Expression
PrintStatement -> Print ( ExpressionList )
Statement -> PrintStatement ;
StatementList -> StatementList Statement
OptStatementList -> StatementList
CompoundStatement -> { OptVariableDeclarationList OptStatementList }
FunctionDefinition -> Type Identifier ( ParameterListOrVoid )
CompoundStatement
Declaration -> FunctionDefinition
DeclarationList -> DeclarationList Declaration
Program -> DeclarationList
```

What additional processing would need to be added for a program with a complex expression like:

```
void main(void) {
    int b;
    int a;
    b = 3;
    a = 12;
    a = (b + 2)-(a*3)/6;
}

main:
        BeginFunc 0     ;
        Var b   ;
        Var a   ;
        Var _t0 ;
        _t0 = 3 ;
        b = _t0 ;
        Var _t1 ;
        _t1 = 12        ;
        a = _t1 ;
        Var _t2 ;
        _t2 = 2 ;
        Var _t3 ;
        _t3 = b + _t2   ;
        Var _t4 ;
        _t4 = 3 ;
        Var _t5 ;
        _t5 = a * _t4   ;
        Var _t6 ;
        _t6 = 6 ;
        Var _t7 ;
        _t7 = _t5 / _t6 ;
        Var _t8 ;
        _t8 = _t3 - _t7 ;
        a = _t8 ;
        EndFunc ;
```

Now let’s consider what needs to be done to deal with arrays:

```
void main(void) {
   int a[2];
   a = NewArray(a);
   a[0] = 1;
   a[1] = 2;
   a[2] = a[1] + a[0];
}

BeginFunc 0     ;
        Var a   ;
        // New array for a
        Var _t0 ;
        Var _t1 ;
        _t1 = 8 ;
        Arg _t1 ;
        _t0 = NewArray()        ;
        a = _t0 ;
        Var _t2 ;
        _t2 = 0 ;
        Var _t3 ;
        Var _t4 ;
        _t4 = 4 ;
        Var _t5 ;
        _t5 = _t4 * _t2 ;
        _t3 = a + _t5   ;
        Var _t6 ;
        _t6 = 1 ;
        *_t3 = _t6      ;
        Var _t7 ;
        _t7 = 1 ;
        Var _t8 ;
        Var _t9 ;
        _t9 = 4 ;
        Var _t10        ;
        _t10 = _t9 * _t7        ;
        _t8 = a + _t10  ;
        Var _t11        ;
        _t11 = 2        ;
        *_t8 = _t11     ;
        Var _t12        ;
        _t12 = 2        ;
        Var _t13        ;
        Var _t14        ;
        _t14 = 4        ;
        Var _t15        ;
        _t15 = _t14 * _t12      ;
        _t13 = a + _t15 ;
        Var _t16        ;
        _t16 = 1        ;
        Var _t17        ;
        Var _t18        ;
        _t18 = 4        ;
        Var _t19        ;
        _t19 = _t18 * _t16      ;
        _t17 = a + _t19 ;
        Var _t20        ;
        _t20 = *_t17    ;
        Var _t21        ;
```

```
                _t21 = 0        ;
                Var _t22        ;
                Var _t23        ;
                _t23 = 4        ;
                Var _t24        ;
                _t24 = _t23 * _t21      ;
                _t22 = a + _t24 ;
                Var _t25        ;
                _t25 = *_t22    ;
                Var _t26        ;
                _t26 = _t20 + _t25      ;
                *_t13 = _t26    ;
                EndFunc ;
```

Before we deal with classes, we should look at how function calls are implemented. This will facilitate our study of methods as they are used in classes. A program with a simple function call (and while we are at it, a global variable too):

```
int a;

int foo(int a, int b) {
   return a + b;
}

void main(void) {
    int b;
    int a;

    foo(a, b);
}
```

Here is the TAC:

```
Var a   ;
_L1:
        BeginFunc 8     ;
        Var a   ;
        Var b   ;
        Var _t0 ;
        _t0 = a + b     ;
        Return _t0      ;
        EndFunc ;
main:
        BeginFunc 0     ;
        Var b   ;
        Var a   ;
        Var _t1 ;
        Arg a   ;
        Arg b   ;
        _t1 = L1()     ;
        EndFunc ;
```

Now for a class example with both fields and methods:

```
class Animal {
  int height;
  void InitAnimal(int h) {
    this.height = h;
  }

  int GetHeight(void) {
    return this.height;
  }
}

class Cow : Animal {
  void InitCow(int h) {
    this.InitAnimal(h);
  }
}

void main(void) {
  class Cow betsy;
  class Animal b;
  betsy = New(Cow);
  betsy.InitCow(5 );
  b = betsy;
  b.GetHeight();
}

_L0:
        BeginFunc 4     ;
        Var h   ;
        Var _t0 ;
        _t0 = This      ;
        Var _t1 ;
        Var _t2 ;
        _t2 = 4 ;
        _t1 = _t0 + _t2 ;
        *_t1 = h        ;
        EndFunc ;
_L1:
        BeginFunc 0     ;
        Var _t3 ;
        _t3 = This      ;
        Var _t4 ;
        Var _t5 ;
        _t5 = 4 ;
        _t4 = _t3 + _t5 ;
        Var _t6 ;
        _t6 = *_t4      ;
        Return _t6      ;
        EndFunc ;
        VTable Animal =
                _L0,
                _L1,
                ;
_L2:
        BeginFunc 4     ;
        Var h   ;
        Var _t7 ;
        _t7 = This      ;
        Arg h   ;
        _t7.0() ;
        EndFunc ;
        VTable Cow =
                _L0,
                _L1,
                _L2,
                ;
main:
        BeginFunc 0     ;
        Var betsy       ;
        Var b   ;
        // New object
        Var _t8 ;
        Var _t9 ;
        _t9 = 8 ;
        Arg _t9 ;
        Arg Cow ;
        _t8 = New()     ;
        betsy = _t8     ;
        Var _t10        ;
        _t10 = 5        ;
        Arg _t10        ;
        betsy.2()       ;
        b = betsy       ;
        Var _t11        ;
        _t11 = b.1()    ;
        EndFunc ;
```

Just two more things to do: if statements and while loops. First, a simple if statement:

```
void main(void) {
  int a;

  a = 23 ;
  if (a == 23)
     a = 10;
  else
     a = 19;
}

main:
        BeginFunc 0     ;
        Var a   ;
        Var _t0 ;
        _t0 = 23        ;
        a = _t0 ;
        Var _t1 ;
        _t1 = 23        ;
        Var _t2 ;
        _t2 = a == _t1  ;
        IfZ _t2 Goto _L0        ;
        Var _t3 ;
        _t3 = 10        ;
        a = _t3 ;
        Goto _L1        ;
_L0:
        Var _t4 ;
        _t4 = 19        ;
        a = _t4 ;
_L1:
EndFunc  ;
```

And finally, a while loop:

```
void main(void) {
  int a;
  a = 0;

  while (a != 10) {
    Print(a, " ");
    a = a + 1;
  }
}


main:
        BeginFunc 0     ;
        Var a   ;
        Var _t0 ;
        _t0 = 0 ;
        a = _t0 ;
_L0:
        Var _t1 ;
        _t1 = 10        ;
        Var _t2 ;
        _t2 = a == _t1  ;
        Var _t3 ;
        _t3 = ! _t2     ;
        IfZ _t3 Goto _L1        ;
        String _L2 = " "        ;
        // Print
        Arg a   ;
        PrintInteger()  ;
        Arg _L2 ;
        PrintString()   ;
        Var _t4 ;
        _t4 = 1 ;
        Var _t5 ;
        _t5 = a + _t4   ;
        a = _t5 ;
        Goto _L0        ;
_L1:
        EndFunc ;
```

Using TAC with other languages

The TAC generation that we have been looking at is fairly generic. Although we have talked about it in the context of SOOP, a TAC generator for any programming language would generate the similar sequence of statements. For example, in the dragon book, the following format is used to define the TAC generation for a while loop. (P. 469 Aho/Sethi/Ullman)

```
S -> while E do S1

{  S.begin := newlabel;
   S.after := newlabel;
   S.code  := gen(S.begin ':') ||
                  E.code ||
                  gen('if' E.place '=' '0'
                         'goto' S.after) ||
                  S1.code ||
                  gen('goto' S.begin) ||
                  gen(S.after ':')
}
```

One last idea before we finish... A nice enhancement to a TAC generator is re-using temp variable names. For example, if we have the following expression:

```
E -> E1 + E2
```

Our usual steps would be to evaluate E1 into t1, evaluate E2 into t2, and then set t3 to their sum. Will t1 and t2 be used anywhere else in the program? How do we know when we can reuse these temp names? Here is a method from Aho/Sethi/Ullman (p. 480) for reusing temp names:

1) Keep a count c initialized to 0.
2) Whenever a temp name is used as an operand, decrement c by 1
3) Whenever a new temp is created, use this new temp and increase c by one.

```
x := a * b + c * d - e * f

(c = 0)  T0 := a * b
(c = 1)  T1 := c * d       (c = 2)
(c = 0)  T0 := T0 + T1
(c = 1)  T1 := e * f       (c = 2)
(c = 0)  T0 := T0 - T1

x := T0
```

Note that this algorithm expects that each temporary name will be assigned and used exactly once, which is true in the majority of cases.

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.