TAC

Handout written by Maggie Johnson and revised by me.

Note: This is an updated version of Handout #26. In the version given out last week, I had missed updating some of the examples to match changes I made to our TAC instruction set. This handout replaces the earlier with correctly updated examples. Recycle the previous version and use this one!

Three address code

Three-address code (TAC) will be the intermediate representation used in our Decaf compiler. It is essentially a 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 operands could be two operands to a binary arithmetic op and the third the result location, or an operand to compare to zero and a second location to branch to, and so on. For example, here is an 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 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 for an if-statement:

```
if (a < b + c)              t1 = b + c;
   a = a - c;               t2 = a < t1;
c = b * c;                  IfZ t2 Goto L0;
                            t3 = a - c;
                            a = t3;
                       L0:  t4 = b * c;
                            c = t4;
```

And here is an example of the translation involving a function call and array access:

```
int n;                      n = LCall _ReadInteger();
n = ReadInteger();          t1 = 4;
Binky(arr[n]);              t2 = t1 * n;
                            t3 = arr + t2;
                            t4 = *(t3);
                            LCall _Binky(t4);
```

Decaf TAC instruction formats

The convention followed in the examples below is that t1, t2, and so on refer to variables (either from source program or temporaries) 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 vtables.

Variable declarations:

```
Var t1;
```

Assignment:

```
t2 = t1;
t1 = "abcdefg";
t1 = 8;
```

(rvalue can be variable, or string/int constant)

Arithmetic:

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

Relational/equality/logical:

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

(not all relational operators are present, must synthesize others using the primitives available)

Labels and branches:

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

(branch test is zero/non-zero)

Function/method calls:

```
LCall L1(t1, t2, t3);
t1 = LCall L1(t1);
```

(different for void vs non-void return)

```
ACall t1(t2, t3);
t0 = ACall t1();
```

(`LCall` used for a function with label known at compile-time, `ACall` for a computed function address, most likely from object vtable)

Function definitions:

```
BeginFuncWithParams t1, t2;
```

(list can be empty if no params)

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

Memory references:

```
t1 = *(t2);
t1 = *(t2 + 8);
*(t1) = t2;
*(t1 + -4) = t2;
```

(optional offset must be integer constant, can be positive or negative)

Array indexing:

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

Object fields, method dispatch:

To access ivars, add offset to base, deref
To call method, retrieve function address from vtable, use `ACall`

Miscellaneous:

```
Halt;
```

(used for runtime errors)

Data specification:

```
VTable ClassName = L1, L2, ...;
```

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

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

   a = 1;
   b = 2;
   c = a + b;
   Print(c);
}
```

```
main:
    BeginFuncWithParams;
    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;
    LCall _PrintInt(c);
    EndFunc;
```

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 create the TAC instructions one at a time. We can immediately print them out or store them for further processing. We will simplify the Decaf language a little by excluding doubles for code generation and internally treating bools as 4-byte integers. Classes, arrays, and strings will be implemented with 4-byte pointers. This means we only ever need to deal with 4-byte integer/pointer variables.

As each production is reduced, we will create the necessary instructions. This strategy makes our code-generation a bit limited—particularly for the way we would have to do switch statements—but we can translate more-or-less any language structure into an executable program in a single pass, without needing to go back and edit anything, which is pretty convenient.

To see how a syntax-directed translation 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 trivial program:

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

```
main:
    BeginFuncWithParams;
    Var _t0;
    _t0 = "hello world";
    LCall _PrintString(_t0);
    EndFunc;
```

Notice that we call the library function labelled _PrintString to do the actual printing. Library functions are called like any ordinary global function, but the code for the definition is provided by the compiler as part of linking with the standard language libraries. Here is the derivation of the source program. The trick is to identify where and what processing occurs as these productions are reduced to generate the given TAC:

```
DeclList ->
Type -> Void
Formals ->
StmtList ->
Constant -> stringConstant
Expr -> Constant
```

```
ExprList-> Expr
PrintStmt -> Print ( ExprList )
Stmt -> PrintStmt ;
StmtList -> StmtList Stmt
StmtBlock -> { StmtList }
FunctionDefn -> Type identifier ( Formals ) StmtBlock
Decl -> FunctionDefn
DeclList -> DeclList Decl
Program -> DeclList
```

Here is another simple program with a bit more complex expression:

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

```
main:
    BeginFuncWithParams;
    Var a;
    Var _t0;
    _t0 = 2;
    Var _t1;
    _t1 = _t0 + a;
    a = _t1;
    LCall _PrintInt(a);
    EndFunc;
```

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

```
DeclList ->
Type -> void
Formals ->
StmtList ->
Type -> int
Variable -> Type identifier
VariableDecl -> Variable ;
Stmt -> VariableDecl
StmtList -> StmtList Stmt
OptReceiver ->
LValue -> OptReceiver identifier
Constant -> intConstant
Expr -> Constant
OptReceiver ->
LValue -> OptReceiver identifier
Expr -> LValue
Expr -> Expr + Expr
SimpleStmt -> LValue = Expr
Stmt -> SimpleStmt ;
StmtList -> StmtList Stmt
OptReceiver ->
LValue -> OptReceiver identifier
Expr -> LValue
ExprList -> Expr
PrintStmt -> Print ( ExprList )
Stmt -> PrintStmt ;
StmtList -> StmtList Stmt
StmtBlock -> { StmtList }
FunctionDefn -> Type identifier ( Formals ) StmtBlock
Decl -> FunctionDefn
DeclList -> DeclList Decl
Program -> DeclList
```

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

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

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

```
main:
    BeginFuncWithParams;
    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 (note the TAC code below doesn't do array bounds checking, that will be your job to implement!)

```
void Binky(int[] arr) {
   arr[1] = arr[0] * 2;
}
```

```
_Binky:
    BeginFuncWithParams arr;
    Var _t0;
    _t0 = 1;
    Var _t1;
    _t1 = 4;
    Var _t2;
    _t2 = _t1 * _t0;
    Var _t3;
    _t3 = arr + _t2;
    Var _t4;
    _t4 = 0;
    Var _t5;
    _t5 = 4;
    Var _t6;
    _t6 = _t5 * _t4;
    Var _t7;
    _t7 = arr + _t6;
    Var _t8;
    _t8 = *(_t7);
    Var _t9;
    _t9 = 2;
    Var _t10;
    _t10 = _t8 * _t9;
    *(_t3) = _t10;
    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:

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

void main() {
    int c;
    int d;

    foo(c, d);
}
```

```
_foo:
    BeginFuncWithParams a, b;
    Var _t0;
    _t0 = a + b;
    Return _t0;
    EndFunc;
main:
    BeginFuncWithParams;
    Var c;
    Var d;
    Var _t1;
    _t1 = LCall _foo(c, d);
    EndFunc;
```

Now for a class example with both fields and methods (notice how `this` is passed as a secret first parameter to a method call)

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

}

class Cow extends Animal {
  void InitCow(int h) {
    InitAnimal(h);
  }
}

void Binky(class Cow betsy) {
  betsy.InitCow(5);
}
```

```
__Animal_InitAnimal:
      BeginFuncWithParams this, h;
      *(this + 4) = h;
      EndFunc;
VTable Animal =
       __Animal_InitAnimal,
;
__Cow_InitCow:
      BeginFuncWithParams this, h;
      Var _t0;
      _t0 = *(this);
      Var _t1;
      _t1 = *(_t0);
      ACall _t1(this, h);
      EndFunc;
 VTable Cow =
       __Animal_InitAnimal,
       __Cow_InitCow,
;
_Binky:
      BeginFuncWithParams betsy;
      Var _t2;
      _t2 = 5;
      Var _t3;
      _t3 = *(betsy);
      Var _t4;
      _t4 = *(_t3 + 4);
      ACall _t4(betsy, _t2);
      EndFunc;
```

How about some TAC that implements control structures, for example, such the if statement below?

```
void main() {
  int a;

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

```
main:
    BeginFuncWithParams;
    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;
```

Or the even snazzier while loop (for loops are left an exercise for the reader):

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

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

```
main:
    BeginFuncWithParams;
    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;
    LCall _PrintInt(a);
    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 Decaf, 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

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.