Three Address Code Examples

Handout written by Maggie Johnson and revised by Julie Zelenski.

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. Some variant of 2, 3 or 4 address code is fairly commonly used as an IR, since it maps well to most assembly languages.

A TAC instruction can have at most three operands. The operands could be two operands to a binary arithmetic operator 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, below on the left is an arithmetic expression and on the right, is a translation into TAC instructions:

```
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 function and method calls. Here is an example of the TAC branching instructions used to translate an if-statement:

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

```
      _t1 = b + c;
      _t2 = a < _t1;
      IfZ _t2 Goto _L0;
      _t3 = a - c;
      a = _t3;
_L0:  _t4 = b * c;
      c = _t4;
```

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

```
n = ReadInteger();
Binky(arr[n]);
```

```
_t0 = LCall _ReadInteger;
n = _t0;
_t1 = 4;
_t2 = _t1 * n;
_t3 = arr + _t2;
_t4 = *(_t3);
PushParam _t4;
LCall _Binky;
PopParams 4;
```

Decaf TAC Instructions

The convention followed in the examples below is that **`t1`**, **`t2`**, and so on refer to variables (either declared variables or temporaries) and **`L1`**, **`L2`**, etc. are used for labels. Labels mark the target for a goto/branch and are used to identify function/method definitions and vtables.

Assignment:

```
t2 = t1;
t1 = "abcdefg";
t1 = 8;
t3 = _L0;
```

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

Arithmetic:

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

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

Relational/equality/logical:

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

(must synthesize other ops as necessary)

Labels and branches:

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

(take branch if value of t1 is zero)

Handling parameters:

```
PushParam t1;
```

(before making call, params are individually pushed right to left)

```
PopParams 24;
```

(after call, pop all params, the number is size in bytes to remove from stack)

Function/method calls:

```
LCall L1;
t1 = LCall L1;
ACall t1;
t0 = ACall t1;
```

(`LCall` a function label known at compile-time, `ACall` a computed function address, most likely from vtable. Each has two forms for void/non-void return value)

Function/method definitions:

```
BeginFunc 12;
```

(the number is size in bytes for all locals and temporaries in stack frame)

```
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 dereference

Object fields, method dispatch:

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

Data specification:

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

Decaf Stanford Library Functions

These functions are invoked using **`LCall`** instructions just like ordinary user-defined functions.

`_Alloc`	one int parameter, returns address of heap-allocated memory of that size in bytes
`_ReadLine`	no parameters, returns string read from user input
`_ReadInteger`	no parameters, returns integer read from user input
`_StringEqual`	two string parameters, returns integer comparison result (neg, 0, pos)
`_PrintInt`	one integer parameter, prints that number to the console
`_PrintString`	one string parameter, prints that string to the console
`_PrintBool`	one boolean parameter, prints true/false to the console
`_Halt`	no parameters, stop program execution

TAC Examples

What we have to do is figure out how to translate from Decaf to TAC. This includes not only generating the TAC, but figuring out the use of temp variables, creating labels, calling functions, etc. As we traverse our tree, we will create the TAC instructions one at a time. We can immediately print them out or store them for further processing. Most of the instructions can be emitted as we go and never re-visited, although in some situations (most notably when performing optimization) we may go back and make changes to earlier instructions based on information we gather later in the process.

We will simplify the Decaf language a little by excluding **`double`**s for code generation and internally treating **`bool`**s 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 we visit each subtree, we will create the necessary instructions. Let's start with a trivial program:

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

```
main:
   BeginFunc 4;
   _t0 = "hello world";
   PushParam _t0;
   LCall _PrintString;
   PopParams 4;
   EndFunc;
```

Visualize the tree for this simple program—there is the program node at the top, its list of declarations has only one entry, the function declaration for **`main`**, within **`main`** we

have a list of statements—in this case, just the one print statement. In order to generate code for a program, we traverse its list of declarations, directing each declaration to generate code. For the **`main`** function, we generate its label and function markers and then iterate over the statements having each emit code. A print statement translates to a call to the Decaf built-in library function **`_PrintString`** to do the actual printing. The library functions are called like any ordinary global function, but the compiler provides the code (and hooks are made the instant the compiler builds the executable.)

Here is another simple program with a little arithmetic:

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

```
main:
    BeginFunc 12;
    _t0 = 2;
    _t1 = _t0 + a;
    a = _t1;
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
```

Consider where the instructions above must be emitted relative to the tree traversal.

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

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

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

```
main:
    BeginFunc 44;
    _t0 = 3;
    b = _t0;
    _t1 = 12;
    a = _t1;
    _t2 = 2;
    _t3 = b + _t2;
    _t4 = 3;
    _t5 = a * _t4;
    _t6 = 6;
    _t7 = _t5 / _t6;
    _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:
    BeginFunc 44;
    _t0 = 1;
    _t1 = 4;
    _t2 = _t1 * _t0;
    _t3 = arr + _t2;
    _t4 = 0;
    _t5 = 4;
    _t6 = _t5 * _t4;
    _t7 = arr + _t6;
    _t8 = *(_t7);
    _t9 = 2;
    _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:
   BeginFunc 4;
   _t0 = a + b;
   Return _t0;
   EndFunc;
main:
    BeginFunc 12;
    PushParam d;
    PushParam c;
    _t1 = LCall _foo;
    PopParams 8;
    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(Cow betsy) {
  betsy.InitCow(5);
}
```

```
_Animal.InitAnimal:
   BeginFunc 0;
      *(this + 4) = h;
      EndFunc;
VTable Animal =
       _Animal.InitAnimal,
;
_Cow.InitCow:
   BeginFunc 8;
      _t0 = *(this);
      _t1 = *(_t0);
      PushParam h;
      PushParam this;
      ACall _t1;
      PopParams 8;
      EndFunc;
 VTable Cow =
       _Animal.InitAnimal,
       _Cow.InitCow,
;
_Binky:
   BeginFunc 12;
      _t2 = 5;
      _t3 = *(betsy);
      _t4 = *(_t3 + 4);
   PushParam _t2;
   PushParam betsy;
      ACall _t4;
   PopParams 8;
      EndFunc;
```

How about some TAC that implements control structures—the **`if`** statement, for example?

```
void main()
{
   int a;

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

```
main:
    BeginFunc 24;
    _t0 = 23;
    a = _t0;
    _t1 = 23;
    _t2 = a == _t1;
    IfZ _t2 Goto _L0;
    _t3 = 10;
    a = _t3;
    Goto _L1;
_L0:
    _t4 = 19;
    a = _t4;
_L1:
    EndFunc;
```

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

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

   while (a < 10) {
      Print(a % 2 == 0);
      a = a + 1;
   }
}
```

```
main:
   BeginFunc 40;
   _t0 = 0;
   a = _t0;
_L0:
   _t1 = 10;
   _t2 = a < _t1;
   IfZ _t2 Goto _L1;
   _t3 = 2;
   _t4 = a % _t3;
   _t5 = 0;
   _t6 = _t4 == _t5;
   PushParam _t6;
   LCall _PrintBool;
   PopParams 4;
   _t7 = 1;
   _t8 = a + _t7;
   a = _t8;
   Goto _L0;
_L1:
   EndFunc;
```

Using TAC For Other Languages

The TAC we use is fairly generic. Although we show our examples in the context of Decaf, a TAC generator for any programming language would generate a similar sequence of statements. For example, in the (in)famous dragon book, the following syntax-directed translation is used to generate TAC for a while loop. (Check out pages 469 of Aho, Sethi, and 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.