Implementing a final code generator

Handout written by Maggie Johnson and revised by me.

The input to the compiler's final code generator is three-address code, hopefully already somewhat optimized. The output from the code generator is an assembly language program specific to some target machine. In order to implement a code generator, we first need to have a clear understanding of the target architecture in terms of register usage, memory organization, instruction scheduling, and so on. We also need to know the instruction set for its assembly language.

You can think of final code generation as just yet another translation task, take input in one format, and produce equivalent output in another. Haven't we tackled tasks like this all quarter? Since our three-address code is a language, we could just write up a grammar for recognizing it. Then, we could add actions to the grammar to emit the appropriate assembly translation as we parse. This is exactly the strategy that Chowder (the code generator we provide for our SOOP compiler) takes. Effectively, it's a compiler inside a compiler— how nifty!

There are two important implementation issues that require some design before embarking on the definition of the grammar and parser:

1) How will registers be allocated?
2) How is global and runtime stack memory used on this machine?

Our approach will be to use a very simple algorithm for register allocation, based on the technique described in lecture. Here is an excerpt from Chowder that shows how we did it:

```
struct Register {
    bool isUsed, isDirty;
    Variable *var;
};

RegisterName RegisterManager::GetRegister(Variable *v, bool writeBeforeRead)
{
     RegisterName reg = FindRegister(v); // find if already in a reg
     if (reg == undef) {
        reg = FindUnusedRegister();          // else find unused one
        if (reg == undef) {
           reg = SelectRegisterToSpill(); // no unused, must spill one
           SpillRegister(reg);
        }
        regs[reg].isUsed = true;         // update register descriptors
        regs[reg].isDirty = false;
        regs[reg].var = v;

  /* load into memory if writeBeforeRead is false; if it's true we will be
     writing over the value in the register so no need to load */
        if (!writeBeforeRead)
           Emit("lw %s, %d(%s)\t\t# load %s", GetNameOfRegister(reg),
                 var->GetOffset(), (var->GetType() == StackBased ?
                 GetNameOfRegister(fp) : GetNameOfRegister(gp)),
                  var->GetName());
     }
     return reg;
}
```

As for memory usage, our TAC generator has already assigned offsets for each of our memory variables, and all this information is stored away in the symbol table. The only thing we have to worry about here is whether we are offsetting from a stack pointer or a global pointer. In the above register code, we can see where this checking is done.

In Chowder, we have a slight problem that one does not normally encounter in a final code generator: we have no symbol table. In a normal compiler, the symbol table that was set up way back in the scanning process persists all the way through to final code generation. But in SOOP, we lost it because Chowder is a stand-alone program that runs after the compiler finishes. So, we need to re-create it. We don't need any of that type/scope information since semantic analysis is long over. All we need is the name and the offset. We do this in a little function called `EmitVar` that is executed any time we find a "Var" in the input:

```
void CodeGenerator::EmitVar(string name, int size) {
  AddVar(name, size);
}
```

which calls:

```
void CodeGenerator::AddVar(string name, int size)
{
  Variable *var;

  if (stackVariables != NULL) {
    var = new Variable(name, StackBased, currentStackOffset);
    currentStackOffset -= size;
    stackVariables->Enter(name, var);
  } else {
    var = new Variable(name, GlobalBased, currentGlobalOffset);
    currentGlobalOffset -= size;
    globalVariables->Enter(name, var);
  }
}
```

As we encounter variables, we assign them offsets (either stack-relative or within the global data segment) and record the name with the offset, so we can look it up later. Because it doesn't have access to a real symbol table, Chowder cuts a few corners, e.g. it won't cope gracefully with shadowed identifiers, so we'll avoid running it on sample programs that show its limitations.

That takes care of the preliminaries. Now we can start thinking about parsing TAC code. (Check handout #26 as a refresher.) What are the tokens in TAC? How hard will it be to write the scanner? What are the valid expressions in TAC? How difficult do you think it will be to write a yacc grammar to recognize the valid constructs? How much error-checking do you think will be required, given that we expect the TAC to be generated by the back-end of our compiler, not by a typo-ridden user?

Here is our version of the Chowder parser, complete with embedded actions:

```
Program:  InstructionList;
InstructionList:  Instruction InstructionList
  |
  ;

Instruction:
     Identifier '=' Identifier BinaryOp Identifier  ';' {
       Emit("# %s = %s %s %s", $1, $3, BinaryOpToString($4), $5);
       CreateVariable(result, $1);
       CreateVariable(left, $3);
       CreateVariable(right, $5);
       gCodeGenerator->EmitBinaryOp(BinaryOpToInstructionName($4), result,
                  left, right);
     }
|  Identifier '=' Identifier  ';' {
       Emit("# %s = %s", $1, $3);
       CreateVariable(result, $1);
       CreateVariable(source, $3);
       gCodeGenerator->EmitCopy(result, source);
     }
|  Identifier '=' Integer ';' {
       Emit("# %s = %d", $1, $3);
       CreateVariable(dest, $1);
       gCodeGenerator->EmitLoadConstant(dest, $3);
     }
  |  Identifier '=' This ';' {
       Emit("# %s = This", $1);
       CreateVariable(dest, $1);
       gCodeGenerator->EmitLoadThis(dest);
     }
  |  Goto Identifier  ';' {
       Emit("# Goto %s", $2);
       gCodeGenerator->EmitGoto($2);
     }
  |  IfZ Identifier Goto Identifier  ';' {
       Emit("# IfZ %s Goto %s", $2, $4);
       CreateVariable(test, $2);
       gCodeGenerator->EmitIfZ(test, $4);
     }
  |  IfNZ Identifier Goto Identifier ';' {
       Emit("# IfZ %s Goto %s", $2, $4);
       CreateVariable(test, $2);
       gCodeGenerator->EmitIfNZ(test, $4);
     }
  |  Arg Identifier ';' {
       Emit("# Arg %s", $2);
       gCodeGenerator->EmitArg($2, 4);
     }
  |  Identifier '=' '*' Identifier ';' {
       Emit("# %s = * %s", $1, $4);
       CreateVariable(result, $1);
       CreateVariable(reference, $4);
       gCodeGenerator->EmitLoadIndirect(result, reference);
     }
  |  '*' Identifier '=' Identifier ';' {
       Emit("# * %s = %s", $2, $4);
```

```
      CreateVariable(value, $4);
      CreateVariable(reference, $2);
      gCodeGenerator->EmitSaveIndirect(reference, value);
    }
|   Identifier '=' Identifier '(' ')' ';' {
      Emit("# %s = %s()", $1, $3);
      CreateVariable(result, $1);
      gCodeGenerator->EmitCall(result, $3);
    }
|   Return Identifier ';' {
      Emit("# Return %s", $2);
      CreateVariable(result, $2);
      gCodeGenerator->EmitReturn(result);
    }
|   Return ';' {
      Emit("# Return");
      gCodeGenerator->EmitReturn(NULL);
    }
|   Identifier ':' {
      gCodeGenerator->EmitLabel($1);
    }
|   Var Identifier ';' {
      Emit("# Var %s", $2);
      gCodeGenerator->EmitVar($2, 4);
    }
|   BeginFunction Integer ';' {
      Emit("# BeginFunc");
      gCodeGenerator->EmitBeginFunction($2);
    }
|   EndFunction ';' {
      Emit("# EndFunc");
      gCodeGenerator->EmitEndFunction();
    }
|   Identifier '.' Integer '(' ')' ';' {
      Emit("# %s.%d()", $1, $3);
      CreateVariable(thisObj, $1);
      gCodeGenerator->EmitMethodCall(NULL, thisObj, $3);
    }
|   Identifier '=' Identifier '.'  Integer '(' ')'  ';' {
      Emit("# %s = %s.%d", $1, $3, $5);
      CreateVariable(result, $1);
      CreateVariable(thisObj, $3);
      gCodeGenerator->EmitMethodCall(result, thisObj, $5);
    }
|   String Identifier '=' StringConstant ';' {
      Emit(".data");
      Emit("%s:", $2);
      Emit(".asciiz %s", $4);
      Emit(".text");
    }
|   VTable Identifier '=' {
      Emit(".data");
      Emit(".align 2");
      Emit("%s:", $2);
    } OptMethodList  ';' {
      Emit(".text");
    }
;
```

```
OptMethodList:
    OptMethodList Identifier ',' {
      Emit(".word %s", $2);
    }
  |
  ;

 BinaryOp:  '*'  { $$ = MultOp; }
  |         '/'   { $$ = DivOp;  }
  |         '-'   { $$ = SubtractOp; }
  |         '+'   { $$ = AddOp; }
  |         '%'   { $$ = ModOp; }
  |        Equal { $$ = EqualOp; }
  |        And   { $$ = AndOp; }
  |        Or    { $$ = OrOp;  }
  |        '<'   { $$ = LessOp; }
  ;

UnaryOp:   '-'  { $$ = NegateOp; }
  |        '!'  { $$ = NotOp;    }
  ;

%%
```

Here is our main program:

```
int main(int argc, string *argv)
{
  gCodeGenerator = new CodeGenerator;
  switch (argc) {
    case 1:
        gCodeGenerator->EmitPreamble();
        yyparse();
        gCodeGenerator->EmitLibrary();
        break;
    case 2:
      if (StringEqual(argv[1], "-n")) {
          gCodeGenerator->EmitPreamble();
          gCodeGenerator->SetSizeVariableName("_lib");
          yyparse();
          return 0;
      }
    default:
      fprintf(stderr, "Usage: chowder <-n> (-n to not link in lib)\n");
  }
}
```

Here are a few interesting functions from codegen.cc:

```
void CodeGenerator::EmitBinaryOp(string name, Variable *result,
             Variable *left, Variable *right)
{
   EmitBinaryInstruction(name,
            registers->GetRegister(result, true),
            registers->GetRegister(left, false),
            registers->GetRegister(right, false));
}
```

```
void CodeGenerator::EmitBinaryInstruction(string name,
                    RegisterName result,
                    RegisterName left,
                    RegisterName right)
{
   Emit("%s %s, %s, %s", name, registers->GetNameOfRegister(result),
            registers->GetNameOfRegister(left),
            registers->GetNameOfRegister(right));
   registers->MarkDirty(result);
}
```

Finally, here is a program with an if statement and our Emit function for generating the marked section of the assembly language:

```
void main(void)
{
   int a;
   if (a == 12)
         a = 1;
   else
         a = 23;
}
```

```
main:
   # BeginFunc
     add $sp, $sp, -12        # space for old ra,fp,this
     sw $fp, 12($sp)          # save prev fp
     sw $ra, 8($sp)           # save prev ra
     sw $a3, 4($sp)           # save prev this
     addiu $fp, $sp, 12       # set up new frame pointer
     move $a3, $a2            # move passed 'this' from a2 to a3
     lw $t0, _lib0
     subu $sp, $sp, $t0       # decrement sp to make space for locals/temps
   # Var a
   # Var _t0
   # _t0 = 12
     li $t0, 12               # load constant value 12 into $t0
   # Var _t1
   # _t1 = a == _t0
     lw $t2, -12($fp)         # load a
     seq $t1, $t2, $t0
   # IfZ _t1 Goto _L0
     sw $t0, -16($fp)         # spill _t0
     sw $t1, -20($fp)         # spill _t1
     beqz $t1, _L0
   # Var _t2
   # _t2 = 1
     li $t0, 1                # load constant value 1 into $t0
   # a = _t2
     move $t1, $t0
   # Goto _L1
     sw $t0, -24($fp)         # spill _t2
     sw $t1, -12($fp)         # spill a
     b _L1

  _L0:
```

```
 # Var _t3
 # _t3 = 23
   li $t0, 23               # load constant value 23 into $t0
 # a = _t3
   move $t1, $t0
   sw $t0, -28($fp)         # spill _t3
   sw $t1, -12($fp)         # spill a
_L1:
 # EndFunc
```

The code generation function that generates the test and branch for an if:

```
void CodeGenerator::EmitIfZ(Variable *test, string label) {
   RegisterName testReg = registers->GetRegister(test, false);
   registers->SpillAllRegisters();
   Emit("beqz %s, %s", registers->GetNameOfRegister(testReg), label);
}
```