Lecture #31: Code Generation

[This lecture adopted in part from notes by R. Bodik]

Updated: This version is modified from the slides in the screencast to conform better to this year's project, and correct a couple of typos.

Intermediate Languages and Machine Languages

- From trees such as output from project #2, could produce machine language directly.
- However, it is often convenient to first generate some kind of *intermediate language (IL):* a "high-level machine language" for a "virtual machine."
- Advantages:
  - Separates problem of extracting the operational meaning (the *dynamic semantics*) of a program from the problem of producing good machine code from it, because it...
  - Gives a clean target for code generation from the AST.
  - By choosing IL judiciously, we can make the conversion of IL $\rightarrow$ machine language easier than the direct conversion of AST $\rightarrow$ machine language. Helpful when we want to target several different architectures (e.g., gcc).
  - Likewise, if we can use the same IL for multiple languages, we can re-use the IL $\rightarrow$ machine language implementation (e.g., gcc, CIL from Microsoft's Common Language Infrastructure).

Stack Machines as Virtual Machines

- A simple evaluation model: instead of registers, a stack of values for intermediate results.
- Examples: The Java Virtual Machine, the Postscript interpreter.
- Each operation (1) pops its operands from the top of the stack, (2) computes the required operation on them, and (3) pushes the result on the stack.
- A program to compute 7 + 5:

```
push 7      # Push constant 7 on stack
push 5
add         # Pop two 5 and 7 from stack, add, and push result.
```

- Advantages
  - Uniform compilation scheme: Each operation takes operands from the same place and puts results in the same place.
  - Fewer explict operands in instructions means smaller encoding of instructions and more compact programs.
  - Meshes nicely with subroutine calling conventions that push arguments on stack.

Stack Machine with Accumulator

- The add instruction does 3 memory operations: Two reads and one write of the stack. The top of the stack is frequently accessed
- Idea: keep most recently computed value in a register (called the *accumulator*) since register accesses are faster.
- For an operation op($e_1, \ldots, e_n$):
  - compute each of $e_1, \ldots, e_{n-1}$ into acc and then push on the stack;
  - compute e_n into the accumulator;
  - perform op computation, with result in acc.
  - pop $e_1, \ldots, e_{n-1}$ off stack.
- The add instruction is now

```
acc := acc + top_of_stack
pop one item off the stack
```

  and uses just one memory operation (popping just means adding constant to stack-pointer register).
- After computing an expression the stack is as it was before computing the operands.

Example: Full computation of 7+5

```
acc := 7
push acc
acc := 5
acc := top_of_stack + acc
pop stack
```


Translating from AST to Stack Machine (I)

- First, it might be useful to have abstractions for our virtual machine and its operations:

```
/** A virtual machine. */
public class VM {
   /** Add INST to our instruction sequence. */
   public void emitInst(Instruction inst);
   ...
}

/** Represents machine instructions in a VM. */
public class Instruction {
   ...
}
```


Translating from AST to Stack Machine (II)

- Let's take a look at a traditional OOP approach in which code generation routines are instance methods in the AST node class.
- A simple recursive pattern usually serves for expressions.
- At the top level, our trees might have an expression-code method:

```
public abstract class Node {
    ...
    /** Generate code for me, leaving my value on the stack. */
    public abstract void cgen(VM machine);
    /** An appropriate VM instruction to use when my operands are on
     *  the stack. */
    abstract Instruction getInst();
    ...
}
```


Translating from AST to Stack Machine (III)

- Implementations of `cgen` then obey this general comment, and each assumes that its children will as well. E.g.

```
public class BinaryExpr extends Node {
   ...
   @Override
   public void cgen(VM machine) {
      left.cgen(machine);
      right.cgen(machine);
      machine.emitInst(getInst());
   }
}
```

- It is up to the implementation of VM to decide how the stack is represented: with all results in memory, or with the most recent in an accumulator.
- Code for `cgen` need not change (example of *separation of concerns*, btw).

The ChocoPy Project Approach

- As you have seen, our projects use a different program structure.
- Functions such as cgen are grouped into analyzers.
- Not really a traditional OOP approach, but it is nice to see the options.
- Here we might write routines such as:

```
public class CodeGenerator extends NodeAnalyzer<Void> {
   public CodeGenerator (VM machine0) {
       machine = machine0;
   }
   ...
   @Override
   public analyze(BinaryExpr node) {
       node.left.dispatch(this);
       node.right.dispatch(this);
       machine.emitInst(node.dispatch(getInstAnalyzer));
       /* I leave getInstAnalyzer to your imagination. */
   }
}
```


From Stack IL to Machine Code (I)

- Eventually, we want to produce machine language.
- To do so, we essentially write another translator from stack language to, say, RISC V.
- This can be simple (and reusable across languages).
- Sample Translation:

```
acc := 7                      li a0, 7
push acc                      addi sp, sp, -4
                              sw a0, 0(sp)
acc := 5                      li a0, 5
acc := top_of_stack + acc     lw t0, 0(sp)
                              add a0, t0, a0
pop stack                     addi sp, sp, 4
```

- As you can see, each statement on the left has a simple translation on the right.
- Unfortunately, there's quite a bit of stack-pointer twiddling going on.

From Stack IL to Machine Code (II)

- An alternative is to allocate all the space needed for the stack (i.e., its maximum in the current function) and keep track of the stack pointer "mentally." (In the project, you can do either, if you choose to use the stack abstraction.)
- Example.

```
Stack                    Previous           Alternative

                                            # At start of function
                                            addi sp, sp, -<size>
...                                         ...
acc := 7                 li a0, 7           li a0, 7
push acc                 addi sp, sp, -4    sw a0, 12(sp) # E.g.
                         sw a0, 0(sp)
acc := 5                 li a0, 5           li a0, 5
acc := top_of_stack + acc lw t0, 0(sp)      lw t0, 12(sp)
                         add a0, t0, a0     add a0, t0, t0
pop stack                addi sp, sp, 4
```


From Stack IL to Machine Code (III)

- So if we had to use several stack slots, we'd simply adjust the immediate offset we use from sp in our code.
- For example, suppose we want to translate x * (a + b):

```
acc := x                      lw a0, x
push acc                      sw a0, 8(sp)  # For example
acc := a                      lw a0, a
push acc                      sw a0, 4(sp)
acc := b                      lw a0, b
acc := top_of_stack + acc     lw t0, 4(sp)
                              add a0, t0, a0
pop stack
acc := top_of_stack * acc     lw t0, 8(sp)
                              mul a0, t0, a0
pop stack
```

- (Alternatively, can use negative offsets from fp as stack offsets, which is what the reference compiler does.)

Virtual Register Machines and Three-Address Code

- Another common kind of virtual machine has an infinite supply of *registers*, each capable of holding a scalar value or address, in addition to ordinary memory.
- A common IL in this case is some form of *three-address code*, so called because the typical "working" instruction has the form

  target := $\text{operand}_1 \oplus \text{operand}_2$

  where there are two source "addresses," one destination "address" and an operation ($\oplus$).
- Often, we require that the operands in the full three-address form denote (virtual) registers or immediate (literal) values, similar to the usual RISC architecture.

Three-Address Code, continued

- A few other forms deal with memory and other kinds of operation:

```
memory_operand := register_or_immediate_operand
register_operand := register_or_immediate_operand
register_operand := memory_operand
goto label
if operand1 ≺ operand2 then goto label
param operand                    ; Push parameter for call.
call operand, # of parameters    ; Call, put return in
                                 ; specific dedicated register
```

- Here, $\prec$ stands for some kind of comparison. Memory operands might be labels of static locations, or indexed operands such as (in C-like notation): `*(r1+4)` or `*(r1+r2)`.

Translating from AST into Three-Address Code

- Change the `cgen` routine to return where it has put its result:

```
public abstract class Node {
    ...
    /** Generate code to compute my value, returning the location
     *  of the result. */
    public Operand cgen(VM machine);
}
```

- Where an Operand denotes some abstract place holding a value.
- Once again, we rely on our children to obey this general comment:

```
public class BinaryExpr extends Callable {
   public Operand cgen(VM machine) {
      Operand leftOp = left.cgen(machine);
      Operand rightOp = right.cgen(machine);
      Operand result = machine.allocateRegister();
      machine.emitInst(result, getInst(), leftOp, rightOp);
      return result;
   }
}
```

- `emitInst` now produces three-address instructions.

A Larger Example

- Consider a small language with integers and integer operations:

```
P:    D ";" P | D
D:    "def" id(ARGS) "=" E;
ARGS: id "," ARGS | id
E:    int | id | "if" E1 "=" E2 "then" E3 "else" E4 "fi"
          | E1 "+" E2 | E1 "-" E2 | id "(" E1,...,En ")"
```

- The first function definition f is the "main" routine
- Running the program on input i means computing f(i)
- Let's continue implementing `cgen` ('+' and '-' already done).

Simple Cases: Literals and Sequences

Conversion of D ";" P:

```
public class StmtList extends Node {
   ...
   public Operand cgen(VM machine) {
      for (int i = 0; i < arity(); i += 1)
         stmts.get(i).cgen(machine);
   }
   return Operand.NoneOperand;
}

public class IntegerLiteral extends Node {
   ...
   @Override
   Operand cgen(VM machine) {
       return machine.immediateOperand(value);
   }
}
```

- `NoneOperand` is an Operand that contains None.

Identifiers

```
public class Identifier : public Node {
   ...
   Operand cgen(VM machine) {
      Operand result = machine.allocateRegister();
      VarInfo info = getInfoFor(name);  // However you do this.
      machine.emitInst(MOVE, result, info.getLocation(machine));
      return result;
   }
}
```

- That is, we assume that the `VarInfo` object that holds information about this occurrence of the identifier contains enough information to get an operand that accesses it from the VM.

Calls

```
public class CallExpr extends Node {
   ...
   @Override
   public Operand cgen(VM machine) {
      for (Node arg : args)
          machine.emitInst(PARAM, arg.cgen(machine));
      Operand callable = function.cgen(machine);
      machine.emitInst(CALL, callable, args.arity());
      return Operand.ReturnOperand;
   }
}
```

- `ReturnOperand` is an abstract location where functions return their value.

Control Expressions: if (Strategy)

- Control expressions generally involve jump and conditional jump instructions.
- To translate

  `if E1 = E2 then E3 else E4 fi`

  we might aim to produce something that realizes the following pseudocode:

```
    code to compute E1 into r1
    code to compute E2 into r2
    if r1 != r2 goto L1
    code to compute E3 into r3
    goto L2
  L1:
    code to compute E4 into r3
  L2:
```

  where the ri denote virtual-machine registers.

Control Expressions: if (Code Generation)

```
public class IfExpr extends Node {
   ...
   public Operand cgen(VM machine) {
      Operand leftOp = left.cgen(machine);
      Operand rightOp = right.cgen(machine);
      Label elseLabel = machine.newLabel();
      Label doneLabel = machine.newLabel();
      machine.emitInst(IFNE, left, right, elseLabel);
      Operand result = machine.allocateRegister();
      machine.emitInst(MOVE, result, thenExpr.cgen(machine));
      machine.emitInst(GOTO, doneLabel);
      machine.placeLabel(elseLabel);
      machine.emitInst(MOVE, result, elseExpr.cgen(machine));
      machine.placeLabel(doneLabel);
      return result;
   }
}
```

- `newLabel` creates a new, undefined instruction label.
- `placeLabel` inserts a definition of the label in the code.

Code generation for 'def'

```
public class FuncDef extends Node {
   ...
   @Override
   Operand cgen(VM machine) {
      machine.placeLabel(name);
      machine.emitFunctionPrologue();
      Operand result = statements.cgen(machine);
      machine.emitInst(MOVE, Operand.ReturnOperand, result);
      machine.emitFunctionEpilogue();
      return Operand.NoneOperand;
   }
}
```

- Where function prologues and epilogues are standard code sequences for entering and leaving functions, setting frame pointers, etc.

A Sample Translation

Program for computing the Fibonacci numbers:

```
def fib(x) = if x = 1 then 0 else
                if x = 2 then 1 else
                   fib(x - 1) + fib(x - 2)
```

Possible code generated:

```
f:  function prologue
    r1 := x
    if r1 != 1 then goto L1
    r2 := 0
    goto L2
L1: r3 := x
    if r3 != 2 then goto L3
    r4 := 1
    goto L4
L3: r5 := x
    r6 := r5 - 1
    param r6
    call fib, 1
    r7 := rret
    r8 := x
    r9 := r8 - 2
    param r9
    call fib, 1
    r10 := r7 + rret
    r4 := r10
L4: r2 := r4
L2: rret := r2
    function epilogue
```