

# Lecture #34: Code Generation

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

# 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 → machine language easier than the direct conversion of AST → 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 → 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 explicit 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  $op(e_1, \dots, e_n)$ :
  - compute each of  $e_1, \dots, e_{n-1}$  into `acc` and then push on the stack;
  - compute  $e_n$  into the accumulator;
  - perform `op` computation, with result in `acc`.
  - pop  $e_1, \dots, e_{n-1}$  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 := acc + top_of_stack
pop stack
```

# A Point of Order

- Often more convenient to push operands in *reverse* order, so right-most operand pushed first.
- This is a common convention for pushing function arguments, and is especially natural when stack grows toward lower addresses.
- Also nice for non-commutative operations on architectures such as the ia32.
- Example: compute  $x - y$ . We show assembly code on the right

<code>acc := y</code>	<code>movl y, %eax</code>
<code>push acc</code>	<code>pushl %eax</code>
<code>acc := x</code>	<code>movl x, %eax</code>
<code>acc := acc - top_of_stack</code>	<code>subl (%esp), %eax</code>
<code>pop stack</code>	<code>addl \$4, %esp</code>

# Translating from AST to Stack Machine

- A simple recursive pattern usually serves for expressions.
- At the top level, our trees might have an expression-code method:

```
class AST {  
    ...  
    /** Generate code for me, leaving my value on the stack. */  
    virtual void cgen (VM* machine);  
}
```

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

```
class BinopNode : public AST {  
    ...  
    void cgen (VM* machine) {  
        getRight ()->cgen (machine);  
        getLeft ()->cgen (machine);  
        machine->emitInst (translateToInst (getOp ()));  
    }  
}
```

We assume here a VM is some abstraction of the virtual machine we're producing code for. `emitInst` adds machine instructions to the program, and `translateToInst` converts, e.g., a '+' to *add*.

# 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

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

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.



## Three-Address Code, continued

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

```
memory_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 r
```

- Here, < 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

- This time, we'll have the `cgen` routine return where it has put its result:

```
class AST {
    ...
    /** Generate code to compute my value, returning the location
     * of the result. */
    virtual 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:

```
class BinopNode : public AST {
    Operand* cgen (VM* machine) {
        Operand* left = getLeft ()->cgen (machine);
        Operand* right = getRight ()->cgen (machine);
        Operand* result = machine->allocateRegister ();
        machine->emitInst (result, translateToInst (getOp ()), left, right);
        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  $f(i)$
- Assume a project-2-like AST.
- Let's continue implementing `cgen` ('+' and '-' already done).

# Simple Cases: Literals and Sequences

## Conversion of D ";" P:

```
class StmtListNode : public AST {
    ...
    Operand* cgen (VM* machine) {
        for (int i = 0; i < arity (); i += 1)
            get (i)->cgen (machine);
    }
    return Operand::NoneOperand;
}
```

```
class IntLiteralNode : public AST {
    ...
    Operand* cgen (VM* machine) {
        return machine->immediateOperand (intTokenValue ());
    }
}
```

- NoneOperand is an Operand that contains None.

# Identifiers

```
class IdNode : public AST {
    ...
    Operand* cgen (VM* machine) {
        Operand result = machine->allocateRegister ();
        machine->emitInst (MOVE, result, getDecl()->getLocation (machine));
        return result;
    }
}
```

- That is, we assume that the declaration object holding information about this occurrence of the identifier contains its location.

# Calls

```
class CallNode : public AST {
    ...
    Operand* cgen (VM* machine) {
        AST* args = getArgList ();
        for (int i = args->arity ()-1; i >= 0; i -= 1)
            machine->emitInst (PARAM, args.get (i)->cgen (machine));
        Operand* callable = getCallable ()->cgen (machine);
        machine->emitInst (CALL, callable, args->arity ());
        return Operand::ReturnOperand;
    }
}
```

- ReturnOperand is abstract location where functions return their value.

# Control Expressions: if

```
class IfExprNode : public AST {
    ...
    Operand* cgen (VM* machine) {
        Operand* left = getLeft ()->cgen (machine);
        Operand* right = getRight ()->cgen (machine);
        Label* elseLabel = machine->newLabel ();
        Label* doneLabel = machine->newLabel ();
        machine->emitInst (IFNE, left, right, elseLabel);
        Operand* result = machine->allocateRegister ();
        machine->emitInst (MOVE, result, getThenPart ()->cgen (machine));
        machine->emitInst (GOTO, doneLabel);
        machine->placeLabel (elseLabel);
        machine->emitInst (MOVE, result, getElsePart ()->cgen (machine));
        machine->placeLabel (doneLabel);
        return result;
    }
}
```

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

# Code generation for 'def'

```
class DefNode : public AST {  
    ...  
    Operand* cgen (VM* machine) {  
        machine->placeLabel (getName ());  
        machine->emitFunctionPrologue ();  
        Operand* result = getBody ()->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*