



COMP 181


Lecture 14
Code generation

November 2, 2005



Prelude

- What is a *qubit*?
 - Quantum bit
- Why quantum computing?
 - “Superposition” can search solutions to a problem simultaneously
 - 3 bits: 1 of 8 possible values
 - 3 qubits = all 8 values, with probabilities
- Is it fundamentally more powerful?
 - No. Just massively parallel.



2

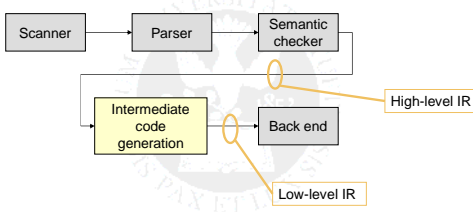

Practical?

3

Today


- Code generation

4

High-level IR

- High-level language constructs
 - Array accesses, field accesses
 - Complex control flow
Loops, conditionals, switch, break, continue
 - Procedures: callers and callees
 - Arithmetic and logic operators
Including things like short-circuit && and ||
- Tree structure
 - Arbitrary nesting of expressions and statements




5

Low-level IR

- Instructions for an abstract machine
- Single operation assignments

$x = y \text{ op } z$

- Operations:
 - Arithmetic: add, sub, etc.
 - Comparisons: >, <, >=, <=
 - Logical operations: &, |, ~
 - Also: unary operations



6

Low-level IR

- Load and store

```
x = *p    x = load p
*p = y    store *p = y
```

- Note: we must build addresses using arithmetic

- Simple control-flow

```
label1:
goto label1
if_goto x, label1
```

Jump to label1
if x has non-zero value



HIR to LIR example

```
if (c == 0) {
  while (c < 20) {
    c = c + 2;
  }
}
else
  c = n * n + 2;
```

```
t1 = c == 0
if_goto t1, lab1
t2 = n * n
c = t2 + 2
goto end
lab1:
t3 = c >= 20
if_goto t3, end
c = c + 2
goto lab1
end:
```



Lowering

- How do we translate from high-level IR to low-level IR?

- HIR is complex, with nested structures
- LIR is low-level, with *everything* explicit
- Need a systematic algorithm

- Idea:

- Define translation for each AST node, *assuming* we have code for children
- Come up with a scheme to stitch them together
- Recursively descend the AST



Lowering scheme

- Define a *generate* function

- For each kind of AST node
- Produces code for that kind of node
- May call generate for children nodes

- How to stitch code together?

- Generate function returns a temporary (or a register) holding the result
- Emit code that combines the results



Lowering expressions

- Arithmetic operations

```
expr1 op expr2
```



```
t1 = generate(expr1)
t2 = generate(expr2)
r = new_temp()
emit( r = t1 op t2 )
return r
```

Generate code for left and right children, get the registers holding the results

Obtain a fresh register name

Emit code for this operation

Return the register to generate call above



Lowering expressions

- Scheme works for:

- Binary arithmetic
- Unary operations
- Logic operations

- What about && and ||?

- In C and Java, they are "short-circuiting"
- Need control flow...

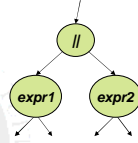


Short-circuiting ||

- If expr1 is true, don't eval expr2

expr1 || expr2

```
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit( r = t1 )
emit( if_goto t1, E )
t2 = generate(expr2)
emit( r = t2 )
emit( E: )
return r
```



Details...

```
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit( r = t1 )
emit( if_goto t1, E )
t2 = generate(expr2)
emit( r = t2 )
emit( E: )
return r
```

```
t1 = expr1
r = t1
if_goto t1, E
t2 = expr2
r = t2
E:
. . .
```

```
t3 =
L:
ifgoto
t1 =
```



Helper functions

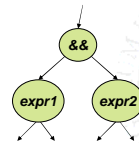
- emit()**
 - The only function that generates instructions
 - Adds instructions to end of buffer
 - At the end, buffer contains code
- new_label()**
 - Generate a unique label name
 - Does not update code
- new_temp()**
 - Generate a unique temporary name
 - May require type information

Order of calls to emit is significant!



Short-circuiting &&

expr1 && expr2



```
N = new_label()
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit( r = t1 )
emit( if_goto t1, N )
emit( goto E )
emit( N: )
t2 = generate(expr2)
emit( r = t2 )
emit( E: )
return r
```



Short-circuiting &&

- Can we do better?

expr1 && expr2



```
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit( r = t1 )
emit( ifnot_goto t1, E )
t2 = generate(expr2)
emit( r = t2 )
emit( E: )
return r
```



Array access

- Depends on abstraction

expr1 [expr2]

```
r = new_temp()
a = generate(expr1)
o = generate(expr2)
emit( o = o * size )
emit( a = a + o )
emit( r = load a )
return r
```

- OR:
 - Emit array op
 - Lower later

Type information from the symbol table



Statements

- Simple sequences


```
statement1;
statement2;
. . .
statementN;
```

```
generate(statement1)
generate(statement2)
. . .
generate(statementN)
```
- Conditionals


```
if (expr)
statement;
```

```
E = new_label()
t = generate(expr)
emit( ifnot_goto t, E )
generate(statement)
emit( E: )
```

Tufts University Computer Science 19

Loops

- Emit label for top of loop
- Generate condition and loop body

```
while (expr)
statement;
```

```
E = new_label()
T = new_label()
emit( T: )
t = generate(expr)
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```

Tufts University Computer Science 20

Function call

- Different calling conventions

```
x = f(expr1,
expr2,
. . .);
```

```
a = generate(f)
foreach expr-1
ti = generate(expr1)
emit( push ti )
emit( call_jump a )
emit( x = get_result )
```

Why call generate here?

Tufts University Computer Science 21

For loop

- How does "for" work?

```
for (expr1; expr2; expr3)
statement
```

Tufts University Computer Science 22

Assignment

- Problem
 - Difference between right-side and left-side
 - Right-side: a value *r-value*
 - Left-side: a location *l-value*
- Example: array assignment


```
A[i] = B[j]
```

Store to this location Load from this location

Tufts University Computer Science 23

Special generate

- Define generate for l-values
 - lgenerate* returns register contains address
 - Simple case: also applies to variables

```
r = new_temp()
a = generate(expr1)
o = generate(expr2)
emit( o = o * size )
emit( a = a + o )
emit( r = load a )
return r
```

```
a = generate(expr1)
o = generate(expr2)
emit( o = o * size )
emit( a = a + o )
return a
```

r-value case *l-value case*

Tufts University Computer Science 24

Assignment

- Use `lgenerate` for left-side
- Return *r-value* for nested assignment

`expr1 = expr2;`

```
r = generate(expr2)
l = lgenerate(expr2)
emit( store *l = r )
return r
```



At leaves

- Depends on level of abstraction
- **generate(v)** – for variables
 - All virtual registers: return v
 - Strict register machine: **emit**(r = load v)
 - Lower level: **emit**(r = load base + offset)
 - “base” is stack pointer, “offset” from symbol table
 - **Note:** may introduces many temporaries
- **generate(c)** – for constants
 - May return special object to avoid r = #



Generation: Big picture

```
Reg generate(ASTNode node)
{
  Reg r;
  switch (node.getKind()) {
  case BIN: t1 = generate(node.getLeft());
            t2 = generate(node.getRight());
            r = new Temp();
            emit( r = t1 op t2 );
            break;
  case NUM: r = new Temp();
            emit( r = node.getValue() );
            break;
  case ID:  r = new Temp();
            o = symtab.getOffset(node.getID());
            emit( r = load sp + o );
            break;
  }
  return r
}
```



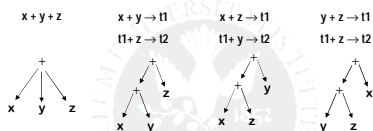
Code Shape

- Definition
 - All those nebulous properties of the code that impact performance & code “quality”
 - Includes:
 - Code for different constructs
 - Cost, storage requirements & mapping
 - Choice of operations
 - Code shape is the end product of many decisions
- Impact
 - Code shape influences algorithm choice & results
 - Code shape can encode important facts, or hide them



Code Shape

- An example:



- What if x is 2 and z is 3?
- What if y+z is evaluated earlier?

Addition is commutative & associative for integers

- The “best” shape for $x+y+z$ depends on context
There may be several conflicting options



Code Shape

- Another example – the switch statement
 - Implement it as a jump table
 - Lookup address in a table & jump to it
 - Uniform (constant) cost
 - Implement it as cascaded if-then-else statements
 - Cost depends on where your case actually occurs
 - $O(\text{number of cases})$
 - Implement it as a binary search
 - Uniform $(\log n)$ cost
- Compiler must choose best implementation strategy
No way to convert one into another



Order of evaluation

- Ordering for performance
 - Using associativity and commutativity
 - Very hard problem
 - Operands
 - op1 must be preserved while op2 is computed
 - Emit code for more intensive one first
- Language requirements
 - Sequence points:
 - Places where side effects must be visible to other operations
 - C examples:

<code>f() + g()</code>	may be executed in any order
<code>f() g()</code>	f must be executed first
<code>f(i++)</code>	argument to f must be i+1



Code Generation

- Tree-walk algorithm
 - Notice: generates code for children first
 - Effectively, a bottom up algorithm
 - So that means....
- Right, syntax directed translation
 - Can emit LIR code in productions
 - Pass registers in \$\$, \$1, \$2, etc.
 - Tricky part: assignment



One-pass code generation

```

Goal: Expr { $$ = $1 }
Expr: Expr + Term
      { r = new_temp();
        emit( r = $1 + $2 );
        $$ = r; }
      | Expr - Term
      { r = new_temp();
        emit( r = $1 - $2 );
        $$ = r; }
      | Term * Fact
      { r = new_temp();
        emit( r = $1 * $2 );
        $$ = r; }
      | Term / Fact
      { r = new_temp();
        emit( r = $1 / $2 );
        $$ = r; }

Fact: ID { r = new_temp();
          o = symtab.getOffset($1);
          emit( r = load sp + o );
          $$ = r; }
      | NUM { r = new_temp();
             emit( r = $1 );
             $$ = r; }
    
```



Next time...

- Generating more efficient code
- Some machine-specific examples

