**Prelude**
- What is this device?  
  - Large Hadron Collider
- What is a hadron?  
  - Subatomic particle made up of quarks bound by the strong force
- What is the premier LHC experiment?  
  - Finding the Higgs boson
- First, what is a boson?  
  - Elementary particle that carries force
- What is the Higgs boson  
  - Interacts with Higgs field  
  - Gives particles mass

**Today**
- Intermediate representations and code generation

```
Theory: Parser  
  - Scanner  
  - Semantic checker  
  - Intermediate code generation
  - Low-level IR  
  - High-level IR  
  - Back end

Examples:  
  - Trees, DAGs  
  - 3 address code  
  - Stack machine code  
  - Control-flow graph
```

**Intermediate representations**
- IR design affects compiler speed and capabilities
- Some important IR properties  
  - Ease of generation, manipulation, optimization  
  - Size of the representation  
  - Level of abstraction: level of “detail” in the IR  
  - How close is IR to source code? To the machine?  
  - What kinds of operations are represented?
- Often, different IRs for different jobs  
  - Typically:  
    - High-level IR: close to the source language  
    - Low-level IR: close to the machine assembly code

**Types of IRs**
- Three major categories  
  - Structural  
    - Graph oriented  
    - Heavily used in IDEs, language translators  
    - Tend to be large
  - Linear  
    - Pseudo-code for an abstract machine  
    - Level of abstraction varies  
    - Simple, compact data structures  
    - Easier to rearrange
  - Hybrid  
    - Combination of graphs and linear code

**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 ||
  - Often: tree structured  
    - Arbitrary nesting of expressions and statements
Abstract Syntax Tree
- AST: parse tree with some intermediate nodes removed

```
*     
x - 2 * y
```
- What is this representation good for?
  - Interpreters
  - We can reconstruct original source
  - Program understanding tools
  - Language translators

Directed Acyclic Graph
- A directed acyclic graph (DAG)
- AST with a unique node for each value

```
z ← x - 2 * y
w ← x / 2
```
- Why do this?
  - More compact (sharing)
  - Encodes redundancy

Low-level IR
- Linear stream of abstract instructions
- Instruction: single operation and assignment

```
x ← y op z
```
- Must break down high-level constructs
  - Example:
    - Introduce temps as necessary: called virtual registers
  - Simple control-flow
    - Label and goto

```
lbl: goto lbl
if goto x, lbl
```

Memory model
- What kind of storage do variables represent?
  - Possibilities:
    - Values on the stack
    - Values in global variables
    - Temporaries introduced during lowering
  - Explicit loads and stores
  - We must compute addresses

```
tl = *p
*t1 = load p
move t1, [p]
```

Stack machines
- Originally for stack-based computers
  - What are advantages?
    - Introduced names are implicit, not explicit
    - Simple to generate and execute code
    - Compact form – who cares about code size?
      - Embedded systems
      - Systems where code is transmitted (the ‘Net)
IR Trade-offs

for (i=0; i<N; i++)
    A[i] = i;

Loop invariant
Strength reduce to temp2 += 4

Towards code generation

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

Lowering scheme

• General scheme
  - Code “template” for each AST node
  - Captures key semantics of each construct
  - Has “holes” for the children of the node
  - Implemented in a function called generate
  - To fill in the template:
    - Call generate function recursively on children
    - Plug code into the holes
  - How to stitch code together?
    - Generate returns a temporary that holds the result
    - Emit code that combines the results

Example: add expression

Source: expr1 + expr2
AST: expr1 + expr2
Flow chart:

Code template:
<code for expr1>
<code for expr2>
<result> = <expr1> + <expr2>
Example: while loop

Source: while (cond) body

AST: while

Flow chart:

Code template:

Generation scheme

- Two problems:
  - Getting the order right
  - How to pass values between the pieces of code

- Solution: order
  - Append each instruction to a global buffer
  - Emit instructions in the desired order

- Solution: passing values
  - Request a new (unique) temporary variable name
  - Generate code that computes value into the temp
  - Return the name of the temp to higher-level generate call

While loop

Compiler:

```java
generate(WhileNode w) {
    E = new_label()
    T = new_label()
    emit( $T: )
    t = generate(w.Condition)
    emit( ifnot_goto $t, $E )
    generate(w.Body)
    emit( goto $T )
    emit( $E: )
}
```

Code template:

Lowering expressions

- Arithmetic operations

```java
expr1 op expr2
```

- Logic operations

```java
expr1 && expr2
```

Lowering scheme

- Emit function
  - Appends low-level abstract instructions to a global buffer
  - Order of calls to emit is important!

- 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

$$\text{expr1 } || \text{ expr2}$$

$E = \text{new_label()}$
$r = \text{new_temp()}$
$t1 = \text{generate(expr1)}$
emit( $r = t1$ )
emit( if goto t1, E )
t2 = \text{generate(expr2)}
emit( $r = t2$ )
emit( E: )
return $r$

Details...

$$E = \text{new_label()}
\begin{array}{l}
  r = \text{new_temp()}
  t1 = \text{generate(expr1)}
  \text{emit( } r = t1 \text{ )}
  \text{emit( if goto t1, E )}
  t2 = \text{generate(expr2)}
  \text{emit( } r = t2 \text{ )}
  \text{emit( E: )}
  \text{return } r
\end{array}$$

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 (from where?)

Short-circuiting $&&$

- Can we do better?

$$\text{expr1 } && \text{ expr2}$$

$E = \text{new_label()}$
$r = \text{new_temp()}$
$t1 = \text{generate(expr1)}
\text{emit( } r = t1 \text{ )}$
\text{emit( ifnot goto t1, E )}$
t2 = \text{generate(expr2)}
\text{emit( } r = t2 \text{ )}$
\text{emit( E: )}$
\text{return } r$

Array access

- Depends on abstraction

$$\text{expr1 [ expr2 ]}$$

$r = \text{new_temp()}
\text{a = generate(expr1)}$
\text{c = generate(expr2)}$
\text{emit( } o = a + c \text{ )}$
\text{emit( } r = \text{load a } \text{ )}$
\text{return } r$

- OR:
  - Emit array op
  - Lower later

Type information from the symbol table
**Statements**

- Simple sequences
  
  ```
  statement1;
  statement2;
  ...
  statementN;
  ```

- Conditionals
  
  ```
  if (expr)
  statement;
  ```

**Loops**

- Emit label for top of loop
- Generate condition and loop body
  
  ```
  while (expr)
  statement:
  ```

**Function call**

- Different calling conventions
  
  ```
  x = f(expr1, expr2, ...
  ```

**For loop**

- How does “for” work?
  
  ```
  for (expr1; expr2; expr3)
  statement
  ```

**Assignment**

- How should we generate `x = y`?
- **Problem**
  
  - Difference between right-side and left-side
  - Right-side: a value
  - Left-side: a location

- Example: array assignment
  
  ```
  A[i] = B[j]
  ```

**Special generate**

- Define special generate for l-values
  
  ```
  r = generate(expr2)
  l = igenerate(expr1)
  emit( store *l = r )
  return r
  ```
Example: arrays

- Two versions of generate:
  ```
  r = new_temp()
  a = generate(arr)
  o = generate(index)
  emit( o = o * size )
  emit( a = a + o )
  emit( r = load a )
  return r
  
  r-value case
  
  a = generate(arr)
  o = generate(index)
  emit( o = o * size )
  emit( a = a + o )
  return a
  
  l-value case
  ```

Example: Big picture
```java
Reg generate(AMYNode 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 <address of o> );
              break;
  }
  return r
}
```
Example

while

\( \text{while} \quad \text{Code} \)

\[ L1: \]

\[ R0 = \text{load } c \]

\[ R1 = 20 \]

\[ R2 = R0 < R1 \]

\[ \text{not}\_\text{goto } R2, L0 \]

\[ E = \text{new}\_\text{label}() = L0 \]

\[ T = \text{new}\_\text{label}() = L1 \]

\[ t = \text{generate}(\text{expr}) \]

\[ =R2 \]

\[ t2 = \text{generate}(\text{expr}) \]

\[ =R1 \]

\[ r = \text{new}\_\text{temp}() \]

\[ =R0 \]

\[ \text{return } r \]

Example

while

\( \text{while} \quad \text{Code} \)

\[ L1: \]

\[ R0 = \text{load } c \]

\[ R1 = 20 \]

\[ R2 = R0 < R1 \]

\[ \text{not}\_\text{goto } R2, L0 \]

\[ E = \text{new}\_\text{label}() = L0 \]

\[ T = \text{new}\_\text{label}() = L1 \]

\[ t = \text{generate}(\text{expr}) \]

\[ =R2 \]

\[ t2 = \text{generate}(\text{expr}) \]

\[ =R1 \]

\[ r = \text{new}\_\text{temp}() \]

\[ =R0 \]

\[ \text{return } r \]
Example

Code

while

Example

Code

Example

Code

Example

Code

while
**Example**

\[
\begin{align*}
&\text{if } c = 0 \\
&\text{not goto R9,L3} \\
L1: & \\
R0 = \text{load } c \\
R1 = 20 \\
R2 = R0 < R1 \\
\text{not goto R2,L0} \\
R3 = \text{load } c \\
R4 = 2 \\
R5 = R3 + R2 \\
\text{store } [R6]=R5 \\
goto L1 \\
L2: & \\
goto L4 \\
L3: & \\
\text{. . .} \\
L4: &
\end{align*}
\]

**Nesting**

\[
\begin{align*}
\text{while } (c < 20) & \\
c & = c + 2 \\
\text{. . .} \\
L4: &
\end{align*}
\]

**Code quality**

- Are there ways to make this code better?

Many CPUs have a fast \(c == 0\) test

Can use accumulators: \(c = c + 2\)

Label leads to another goto; may have multiple labels

**Efficient lowering**

- Reduce number of temporary registers
- Don’t copy variable values unnecessarily
- Accumulate values, when possible
- Reuse temporaries, where possible
- Generate more efficient labels
- Don’t generate multiple adjacent labels
- Avoid goto-label-goto
- Typically done later, as a separate control-flow optimization

**Registers only**

- RISC-like low-level IR

In practice:
  - Assume all variables are in registers
  - Add loads and stores later as part of register allocation

**Avoiding extra copies**

- Basic algorithm
- Recursive generation traverses to leaves
- At leaves, generate: \(R = v\) or \(R = c\)

Improvement
- Stop recursion one level early
- Check to see if children are leaves
- Don’t call generate recursively on variables, constants
Avoiding copies

\[ \text{expr1 op expr2} \]

if (expr1 is a Var)
   \( t_1 = \text{Var} \) \( \text{expr1} \)
else
   \( t_1 = \text{generate(} \text{expr1} \text{)} \)
if (expr2 is a Var)
   \( t_2 = \text{Var} \) \( \text{expr2} \)
else
   \( t_2 = \text{generate(} \text{expr2} \text{)} \)
\( r = \text{new_temp}() \)
\( \text{emit( } r = t_1 \text{ op } t_2 \text{ )} \)
return \( r \)

Example

- Expr1 is (a+b)
  - Not a leaf
  - Recursively generate code
  - Return temp
- Expr2 is c
  - Return c
- Emit \( (R_0 \times c) \)

Use accumulation

- Idea:
  - We only need 2 registers to evaluate \( \text{expr1 op expr2} \)
  - Reuse temp assigned to one of the subexpressions

if (expr1 is \text{var})
   \( t_1 = \text{Var} \) \( \text{expr1} \)
else
   \( t_1 = \text{generate(} \text{expr1} \text{)} \)
if (expr2 is \text{var})
   \( t_2 = \text{Var} \) \( \text{expr2} \)
else
   \( t_2 = \text{generate(} \text{expr2} \text{)} \)
\( r = \text{new_temp}() \)
\( \text{emit( } r = t_1 \text{ op } t_2 \text{ )} \)
return \( t_1 \)

Example

- Combined:
  - Remove copies
  - Accumulate value
  - Only need one register
  - How many would the original scheme have used?

\( (a + b) \times c \)

Code

\( R_0 = a + b \)
\( R_0 = R_0 \times c \)

Reuse of temporaries

- Idea:
  - Can \( \text{generate(} \text{expr1} \text{)} \) and \( \text{generate(} \text{expr2} \text{)} \) share temporaries?
    - Yes, except for \( t_1 \) and \( t_2 \)
    - Observation: temporaries have a limited lifetime
    - Lifetime confined to a subtree

\[ \text{expr1 op expr2} \]

\( t_1 = \text{generate(} \text{expr1} \text{)} \)
\( t_2 = \text{generate(} \text{expr2} \text{)} \)
\( r = \text{new_temp}() \)
\( \text{emit( } r = t_1 \text{ op } t_2 \text{ )} \)
return \( r \)

Example

- Subtrees can share registers

- Algorithm:
  - Use a stack of registers
  - Start at \# = 0
  - Each call to generate:
    - "Push" next number
    - Use any register > \# (if available)
    - When done, "pop" back up

\( R_n \times c \)

Code

\( R_0 = a + b \)
\( R_0 = R_0 \times c \)
\( R_0 = R_0 \times c \)
Miscellaneous

- Code “shape”
  - Consider expression: \( x + y + z \)
  - Code:
    
    \[
    \begin{align*}
    t1 &= x + y \\
    t2 &= t1 + z \\
    t3 &= y + z \\
    t4 &= t2 + t3 \\
    \end{align*}
    \]
  - What if \( x = 3 \) and \( y = 2 \)?
  - What if \( y + z \) evaluated earlier in code?

- Ordering for performance
  - Using associativity and commutativity – very hard
  - Operands
    - \( op1 \) must be preserved while \( op2 \) is computed
    - Emit code for more intensive one first

One-pass code generation

```
Goal ::= Expr {: RES = \_ ;}
Expr ::= Expr + Term {
  \_ = new_temp();
  emit( \_ = e + t );
  RES = \_ ;}
| Expr - Term {
  \_ = new_temp();
  emit( \_ = e - t );
  RES = \_ ;}

Term ::= Term * Fact {
  \_ = new_temp();
  emit( \_ = t * f );
  RES = \_ ;}
| Term / Fact {
  \_ = new_temp();
  emit( \_ = t / f );
  RES = \_ ;}

Fact ::= ID {
  \_ = new_temp();
  emit( \_ = load <address of \_> );
  RES = \_ ;}
| NUM {
  \_ = new_temp();
  emit( \_ = \$n );
  RES = \_ ;}
```

Code Generation

- Tree-walk algorithm
  - Notice: generates code for children first
  - Effectively, a bottom up algorithm
  - So that means….

- Right! Use syntax directed translation
  - Can emit LIR code in productions
  - Pass register names in \$\$, \$1\$, \$2\$, etc.
  - Can generate assembly: one-pass compiler
  - Tricky part: assignment

Next time

- Study the IRs of your compiler
- Implementing procedures and methods
- New programming assignment