Code Generation

```plaintext
loadI 3 => r1
loadI 4 => r2
mult r1, r2 => r3
loadI 2 => r4
add r3, r4 => r5
print r5
```
Compilers

Source code

Lexing

"Front end"

Tokens

Parsing

Syntax tree

"Back end"

Code Generation & Optimization

Machine code

Current focus
Our Project

- Current status: type-checked AST
- Next step: convert to ILOC
  - This step is called *code generation*
  - Convert from a tree-based IR to a linear IR
    - Or directly to machine code (uncommon)
    - Use a tree traversal to “linearize” the program
Goals

- **Linear codes**
  - **Stack code** \((\text{push } a, \text{ push } b, \text{ multiply}, \text{ pop } c)\)
  - **Three-address code** \((c = a + b)\)
  - **Machine code** \((\text{movq } a, %eax; \text{ addq } b, %eax; \text{ movq } %eax, c)\)

- **Code generator requirements**
  - Must preserve semantics
  - Should produce efficient code
  - Should run efficiently
Obstacles

• Generating the most optimal code is undecidable
  – Unlike front-end transformations
    • (e.g., lexing & parsing)
  – Must use heuristics and approximation algorithms
  – This is why most compilers research since 1960s has been on the back end
ILOC

- Linear IR based on research compiler from Rice
- See Appendix A (and ILOCInstruction / ILOCInterpreter)
- I have made some modifications to simplify P5
  - Removed most immediate instructions (i.e., \texttt{subI})
  - Removed binary shift instructions
  - Removed character-based instructions
  - Removed jump tables
  - Removed comparison-based conditional jumps
  - Added labels and function call mechanisms (\texttt{call}, \texttt{param}, \texttt{return})
  - Added binary \texttt{not} and arithmetic \texttt{neg}
  - Added \texttt{print} and \texttt{nop} instructions
• Simple von Neumann architecture
  - Not an actual hardware architecture, but useful for teaching
  - 32-bit words w/ 64K address space
  - Read-only code region indexed by instruction
  - Unlimited 32-bit integer virtual registers (r1, r2, …)
  - Four special-purpose registers:
    • IP: instruction pointer
    • SP: stack pointer
    • BP: base pointer
    • RET: return value
<table>
<thead>
<tr>
<th>Form</th>
<th>Op1</th>
<th>Op2</th>
<th>Op3</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>op1, op2 =&gt;</td>
<td>reg</td>
<td>reg</td>
<td>op3</td>
</tr>
<tr>
<td>sub</td>
<td>op1, op2 =&gt;</td>
<td>reg</td>
<td>reg</td>
<td>op3</td>
</tr>
<tr>
<td></td>
<td>addition</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td>op1, op2 =&gt; op3</td>
<td>reg</td>
<td>reg</td>
<td>multiplication w/ constant</td>
</tr>
<tr>
<td>div</td>
<td>op1, op2 =&gt; op3</td>
<td>reg</td>
<td>reg</td>
<td>division</td>
</tr>
<tr>
<td>addI</td>
<td>op1, op2 =&gt; op3</td>
<td>reg</td>
<td>imm</td>
<td>reg</td>
</tr>
<tr>
<td>multI</td>
<td>op1, op2 =&gt; op3</td>
<td>reg</td>
<td>imm</td>
<td>multiplication w/ constant</td>
</tr>
<tr>
<td>neg</td>
<td>op1         =&gt; op2</td>
<td>reg</td>
<td>reg</td>
<td>arithmetic negation</td>
</tr>
</tbody>
</table>

**Integer Arithmetic**

| and   | op1, op2 => op3 | reg         | reg         | reg                              |
| or    | op1, op2 => op3 | reg         | reg         | reg                              |
| not   | op1         => op2 | reg         | reg         | reg                              |

**Boolean Arithmetic**

| i2i   | op1         => op2 | reg         | reg         | register copy                    |
| loadI | op1         => op2 | imm         | reg         | load integer constant            |
| loadS | &op1        => op2 | sym         | reg         | load symbol address              |
| load  | [op1]       => op2 | reg         | reg         | load from address                |
| loadAI| [op1+op2]   => op3 | reg         | imm         | reg                              |
| loadAO| [op1+op2]   => op3 | reg         | reg         | reg                              |
| store | op1         => [op2] | reg         | reg         | store to address                 |
| storeAI| op1        => [op2+op3] | reg         | reg         | reg                              |
| storeAO| op1        => [op2+op3] | reg         | reg         | reg                              |
## Comparison

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp_LT op1, op2 =&gt; op3</td>
<td>reg, reg</td>
<td>less-than comparison</td>
</tr>
<tr>
<td>cmp_LE op1, op2 =&gt; op3</td>
<td>reg, reg</td>
<td>less-than-or-equal-to comparison</td>
</tr>
<tr>
<td>cmp_EQ op1, op2 =&gt; op3</td>
<td>reg, reg, reg</td>
<td>equality comparison</td>
</tr>
<tr>
<td>cmp_GE op1, op2 =&gt; op3</td>
<td>reg, reg, reg</td>
<td>greater-than-or-equal-to comparison</td>
</tr>
<tr>
<td>cmp_GT op1, op2 =&gt; op3</td>
<td>reg, reg, reg</td>
<td>greater-than comparison</td>
</tr>
<tr>
<td>cmp_NE op1, op2 =&gt; op3</td>
<td>reg, reg, reg</td>
<td>inequality comparison</td>
</tr>
</tbody>
</table>

## Control Flow

<table>
<thead>
<tr>
<th>Command</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td></td>
<td>control flow label</td>
</tr>
<tr>
<td>jump</td>
<td>op1</td>
<td>unconditional branch</td>
</tr>
<tr>
<td>cbr</td>
<td>op1 =&gt; op2, op3</td>
<td>conditional branch</td>
</tr>
<tr>
<td>param</td>
<td>op1</td>
<td>pass parameter</td>
</tr>
<tr>
<td>call</td>
<td></td>
<td>call function</td>
</tr>
<tr>
<td>return</td>
<td></td>
<td>return to caller</td>
</tr>
</tbody>
</table>

## Miscellaneous

<table>
<thead>
<tr>
<th>Command</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>print</td>
<td></td>
<td>print value to standard out</td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td>no-op (do nothing)</td>
</tr>
<tr>
<td>phi</td>
<td>reg, reg</td>
<td>φ-function (for SSA only)</td>
</tr>
</tbody>
</table>
Syntax-Directed Translation

- Similar to attribute grammars (Figure 4.15)
- Create code-gen routine for each production
  - Each routine generates code based on a template
  - Save intermediate results in temporary registers
- In our project, we will use a visitor
  - Still syntax-based (actually AST-based)
  - Not dependent on original grammar
  - Generate code as a synthesized attribute ("code")
  - Save temporary registers as another attribute ("reg")
Sample code:

```
loadI 3 => r1
loadI 4 => r2
mult r1, r2 => r3
loadI 2 => r4
add r3, r4 => r5
print r5
```

Decaf equivalent:

```
print_int(2+3*4);
```
• Sample code:

    loadI 3 => r1
    loadI 4 => r2
    mult r1, r2 => r3
    loadI 2 => r4
    add r3, r4 => r5
    print r5

    Decaf equivalent:
    print_int(2+3*4);

    loadI 3 => r1
    loadI 4 => r2
    mult r1, r2 => r3
    loadI 2 => r4
    add r3, r4 => r5
    print r5
Sample code:

```assembly
loadI 5 => r1
loadI 8 => r2
add r1, r2 => r3
loadI 10 => r4
cmp_LT r3, r4 => r5
cbr r5 => l1, l2

l1:
    print "yes"
jmp l3
l2:
    print "no"
l3:
```

Decaf equivalent:

```java
if (5 + 8 < 10) {
    print_str("yes");
} else {
    print_str("no");
}
```
Boolean Encoding

• Integers: 0 for false, 1 for true
• Difference from book
  – No comparison-based conditional branches
  – Conditional branching uses boolean values instead
  – This enables simpler code generation
• Short-circuiting
  – Not in Decaf!
String Handling

• Arrays of chars vs. encapsulated type
  – Former is faster, latter is easier/safer
  – C uses the former, Java uses the latter

• **Mutable vs. immutable**
  – Former is more intuitive, latter is (sometimes) faster
  – C uses the former, Java uses the latter

• Decaf: immutable string constants only
  – No string variables
Array Accesses

• 1-dimensional case: \( \text{base} + \text{width} \times i \)

• Generalization for multiple dimensions:
  - \( \text{base} + (i_1 \times w_1) + (i_2 \times w_2) + \ldots + (i_k \times w_k) \)

• Alternate definition:
  - 1d: \( \text{base} + \text{width} \times (i_1) \)
  - 2d: \( \text{base} + \text{width} \times (i_1 \times n_2 + i_2) \)
  - nd: \( \text{base} + \text{width} \times ((( \ldots ((i_1 \times n_2 + i_2) \times n_3 + i_3) \ldots ) \times n_k + i_k) \times \text{width} \)

• **Row-major vs. column-major**

• In Decaf: row-major one-dimensional global arrays
Struct and Record Types

• How to access member values?
  - Static offsets from base of struct/record

• OO adds another level of complexity
  - Now classes have methods
  - Class instance records and virtual method tables

• In Decaf: no structs or classes
Control Flow

• Introduce labels
  – Named locations in the program
  – Generated sequentially using static `newlabel()` call

• Generate jumps/branches using templates
  – In ILOC: “cbr” instruction (no fallthrough!)
  – Templates are composable
if statement: if \((E)\) B1

\[
\begin{align*}
\text{rE} &= <<\ E\ \text{code} >> \\
cbr\ \text{rE} &\rightarrow b1, \text{skip} \\
b1: & \\
&<<\ B1\ \text{code} >> \\
skip: &
\end{align*}
\]
if statement: if (E) B1 else B2

\[ rE = \text{<< E code >>} \]
\[ \text{cbr } rE \rightarrow b1, b2 \]

b1:
\[ \text{<< B1 code >>} \]
\[ \text{jmp done} \]

b2:
\[ \text{<< B2 code >>} \]

done:
Control Flow

while loop: while (E) B
while loop: **while (E) B**

**cond:**

\[ rE = \langle\langle E \text{ code} \rangle\rangle \]

cbr \( rE \rightarrow \) body, done

**body:**

\[ \langle\langle B \text{ code} \rangle\rangle \]

jmp cond

**done:**
while loop: **while (E) B**

cond:  
\[ rE = \ll E \text{ code } \gg \]
\[ \text{cbr} \ rE \rightarrow \text{body, done} \]

body:
\[ \ll B \text{ code } \gg \]
\[ \text{jmp} \ \text{cond} \]

done:  
\[ ; \text{BREAK target} \]
for loop: for V in E1, E2 B

rX = << E1 code >>
rY = << E2 code >>
rV = rX

cond:
    cmp_GE rV, rY → rC
cbr rC → done, body

body:
    << B code >>
rV = rV + 1 ; CONTINUE target
    jmp cond

done: ; BREAK target
- Static single-assignment
  - Unique name for each newly-calculated value
  - Values are collapsed at control flow points using Φ-functions
    - (not actual executed!)
  - Useful for various types of analysis
  - We’ll generate ILOC in SSA for P5

```c
if (a < b) {
    c = 4;
} else {
    c = 8;
}
```
Procedure Calls

• These are harder
  – We'll talk about them next week