```plaintext
loadI 3 => r1
loadI 4 => r2
mult r1, r2 => r3
loadI 2 => r4
add r3, r4 => r5
print r5
```
char data[20];
int main()
{
    float x;
    x = 42.0;
    return 7;
}
Compilers

- 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 (push a, push b, multiply, pop c)
  - Three-address code (c = a + b)
  - Machine code (movq a, %eax; addq b, %eax; movq %eax, c)

- Code generator requirements
  - Must preserve semantics
  - Should produce efficient code
  - Should run efficiently
Generating optimal code is undecidable

- Unlike front-end transformations
  - (e.g., lexing & parsing)
- Must use heuristics and approximation algorithms
  - **Systems design involves trade-offs** (e.g., speed for code size)
  - Sometimes “best” choice depends on target architecture (ISA, cache sizes, etc.)
- 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., subI)
  - Removed binary shift instructions
  - Removed character-based instructions
  - Removed jump tables
  - Removed comparison-based conditional jumps
  - Added stack operations push and pop
  - Added labels and function call instructions call and return
  - Added binary not and arithmetic neg
  - Added print and 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></td>
<td></td>
<td></td>
<td></td>
<td><strong>Integer Arithmetic</strong></td>
</tr>
<tr>
<td>add</td>
<td>op1</td>
<td>op2</td>
<td>op3</td>
<td>reg reg reg</td>
</tr>
<tr>
<td>sub</td>
<td>op1</td>
<td>op2</td>
<td>op3</td>
<td>reg reg reg</td>
</tr>
<tr>
<td>mult</td>
<td>op1</td>
<td>op2</td>
<td>op3</td>
<td>reg reg reg</td>
</tr>
<tr>
<td>div</td>
<td>op1</td>
<td>op2</td>
<td>op3</td>
<td>reg reg reg</td>
</tr>
<tr>
<td>addI</td>
<td>op1</td>
<td>op2</td>
<td>op3</td>
<td>reg imm reg</td>
</tr>
<tr>
<td>multI</td>
<td>op1</td>
<td>op2</td>
<td>op3</td>
<td>reg imm reg</td>
</tr>
<tr>
<td>neg</td>
<td>op1</td>
<td></td>
<td>op2</td>
<td>reg reg</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Boolean Arithmetic</strong></td>
</tr>
<tr>
<td>and</td>
<td>op1</td>
<td>op2</td>
<td>op3</td>
<td>reg reg reg</td>
</tr>
<tr>
<td>or</td>
<td>op1</td>
<td>op2</td>
<td>op3</td>
<td>reg reg reg</td>
</tr>
<tr>
<td>not</td>
<td>op1</td>
<td></td>
<td>op2</td>
<td>reg reg</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Data Movement</strong></td>
</tr>
<tr>
<td>i2i</td>
<td>op1</td>
<td></td>
<td>op2</td>
<td>reg reg</td>
</tr>
<tr>
<td>loadI</td>
<td>op1</td>
<td></td>
<td>op2</td>
<td>imm reg</td>
</tr>
<tr>
<td>load</td>
<td>[op1]</td>
<td></td>
<td>op2</td>
<td>reg reg</td>
</tr>
<tr>
<td>loadAI</td>
<td>[op1+op2]</td>
<td></td>
<td>op3</td>
<td>reg imm reg</td>
</tr>
<tr>
<td>loadAO</td>
<td>[op1+op2]</td>
<td></td>
<td>op3</td>
<td>reg reg reg</td>
</tr>
<tr>
<td>store</td>
<td>op1</td>
<td>=&gt;</td>
<td>[op2]</td>
<td>reg reg</td>
</tr>
<tr>
<td>storeAI</td>
<td>op1</td>
<td>=&gt;</td>
<td>[op2+op3]</td>
<td>reg reg imm</td>
</tr>
<tr>
<td>storeAO</td>
<td>op1</td>
<td>=&gt;</td>
<td>[op2+op3]</td>
<td>reg reg reg</td>
</tr>
<tr>
<td>push</td>
<td>op1</td>
<td>=&gt;</td>
<td></td>
<td>reg</td>
</tr>
<tr>
<td>pop</td>
<td>op1</td>
<td></td>
<td></td>
<td>reg</td>
</tr>
</tbody>
</table>
### Comparison

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp_LT</td>
<td>less-than comparison</td>
</tr>
<tr>
<td>cmp_LE</td>
<td>less-than-or-equal-to comparison</td>
</tr>
<tr>
<td>cmp_EQ</td>
<td>equality comparison</td>
</tr>
<tr>
<td>cmp_GE</td>
<td>greater-than-or-equal-to comparison</td>
</tr>
<tr>
<td>cmp_GT</td>
<td>greater-than comparison</td>
</tr>
<tr>
<td>cmp_NE</td>
<td>inequality comparison</td>
</tr>
</tbody>
</table>

### Control Flow

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>control flow label</td>
</tr>
<tr>
<td>jump</td>
<td>unconditional branch</td>
</tr>
<tr>
<td>cbr</td>
<td>conditional branch</td>
</tr>
<tr>
<td>call</td>
<td>call function</td>
</tr>
<tr>
<td>return</td>
<td>return to caller</td>
</tr>
</tbody>
</table>

### Miscellaneous

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>print</td>
<td>print integer to standard out</td>
</tr>
<tr>
<td>print</td>
<td>print string to standard out</td>
</tr>
<tr>
<td>nop</td>
<td>no-op (do nothing)</td>
</tr>
<tr>
<td>phi</td>
<td>ϕ-function (for SSA only)</td>
</tr>
</tbody>
</table>

### x86-64:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpq %r2, %r1</td>
<td>ILOC:</td>
</tr>
<tr>
<td>jl L1</td>
<td>cmp_LT r1, r2 =&gt; rE</td>
</tr>
<tr>
<td>jmp L2</td>
<td>cbr rE =&gt; L1, L2</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")
  - **Operational semantics** rules describe this process formally

\[
\text{SBlock} \quad \frac{s_1 \rightarrow C_1 \quad s_2 \rightarrow C_2 \quad ... \quad s_n \rightarrow C_n}{\{s_1, s_2, ..., s_n\} \rightarrow C_1; \ C_2; \ ... \ C_n}
\]
Sample code:

```plaintext
print_int(2+3*4);

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

Decaf equivalent:

```plaintext
print_int(2+3*4);
```
Example

- Sample code:

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

Decaf equivalent:

```
print_int(2+3*4);
```

```
// ASTLiteral (2)
// ASTLiteral (3)
// ASTLiteral (4)
// ASTBinaryExpr (*)
// ASTBinaryExpr (+)
// ASTVoidFunctionCall (print_str)
```

```
BinEx (+)

Lit(2)      BinEx (*)

Lit(3)      Lit(4)
```

```
SInt
INT  \rightarrow  \langle\text{loadI INT}  \Rightarrow  r, r\rangle
```

```
SAdd
\begin{align*}
  e_1 \rightarrow  \langle C_1, r_1 \rangle  & \quad e_2 \rightarrow  \langle C_2, r_2 \rangle \\
  e_1 \, '+' \, e_2 \rightarrow  \langle C_1; C_2; \text{add } r_1, r_2  \Rightarrow  r_3, r_3 \rangle
\end{align*}
```

(similar for SSub (-), SMul (*), SDiv (/), SAnd (&&), and SOr (||))
**Example**

2 + 3 * 4

**Code:**
- `loadI 2 => r1`
- `loadI 3 => r2`
- `loadI 4 => r3`
- `mult r2, r3 => r4`
- `add r1, r4 => r5`

**Reg:** r5

**Code:**
- `loadI 3 => r2`
- `loadI 4 => r3`
- `mult r2, r3 => r4`

**Reg:** r4

**Code:**
- `loadI 2 => r1`

**Reg:** r1
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!

\[
\begin{align*}
\text{STrue} & : \text{true} \rightarrow \langle \text{loadI 1} \Rightarrow r, r \rangle \\
\text{SFalse} & : \text{false} \rightarrow \langle \text{loadI 0} \Rightarrow r, r \rangle
\end{align*}
\]
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
Variables

• Global: access using static address
  – Load into temporary base register first (no offset)
• Local: access using offset from base pointer (BP)
  – For ILOC, 4-byte slots starting at \([bp-4]\) (so \([bp-8]\), \([bp-12]\), etc.)
  – Assume we can look up base register and constant offset

```c
int a; int b; int c;
...
c = a + b;
```

\[ \text{loadAI} \ [bp-4] \Rightarrow r1 \]
\[ \text{loadAI} \ [bp-8] \Rightarrow r2 \]
\[ \text{add} \ r1, \ r2 \Rightarrow r3 \]
\[ \text{storeAI} \ r3 \Rightarrow [bp-12] \]

\[ \text{SLoc} \quad r_b = \text{base}(\text{ID}) \quad x_o = \text{offset}(\text{ID}) \implies \text{ID} \rightarrow \langle \text{loadAI} [r_b + x_o] \Rightarrow r, r \rangle \]

\[ \text{SAssign} \quad e \rightarrow \langle C_e, r_e \rangle \quad r_b = \text{base}(\text{ID}) \quad x_o = \text{offset}(\text{ID}) \implies \text{ID} = e \rightarrow C_e; \text{storeAI} \ r_e \Rightarrow [r_b + x_o] \]
Array Accesses

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

• Generalization for multiple dimensions:
  \[ \text{base} + (i_1 \times n_1) + (i_2 \times n_2) + \ldots + (i_k \times n_k) \]

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

• Row-major vs. column-major

• In Decaf: row-major one-dimensional global arrays

\[
\begin{align*}
\text{SArrLoc} & \quad e \rightarrow \langle C_e, r_e \rangle \\
\text{ID}[e] & \rightarrow \langle C_e; \text{multi} \ r_e, x_s \Rightarrow r_o; \text{loadA} \ [r_b+r_o] \Rightarrow r, r \rangle
\end{align*}
\]
Struct and Record Types

- Access fields using static offsets from base of struct
- OO adds another level of complexity
  - Must include storage for inherited fields
  - Must handle dynamic dispatch for method calls
  - 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 code templates
  - Similar to `do-while`, `jump-to-middle`, and `guarded-do` from CS 261
  - In ILOC: “cbr” instruction (no fallthrough!)
    - So the CS 261 templates won’t work verbatim
  - Templates are composable
  - **Operational semantics** rules describe these templates
Control Flow

if statement: \textbf{if} (E) \textbf{B1}

\[
\begin{align*}
\text{rE} &= \text{<< E code >>} \\
\text{cbr rE} &\rightarrow b1, \text{ skip} \\
b1: & \quad \text{<< B1 code >>} \\
\text{skip:}
\end{align*}
\]

\[
\text{SLIf} \quad e \rightarrow \langle C_e, r_e \rangle \quad b \rightarrow C_b
\]

| if '(' e ')', b \rightarrow C_e; cbr r_e \Rightarrow l_1, l_2; l_1:: C_b; l_2: |
Control Flow

if statement: \textbf{if (E) B1 else B2}

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

\begin{align*}
\text{b1:} & \\
& \text{<< B1 code >>} \\
& \text{jmp done} \\
\text{b2:} & \\
& \text{<< B2 code >>} \\
\text{done:} & 
\end{align*}

\[
\text{SIIfElse} \quad e \rightarrow \langle C_e, r_e \rangle \quad b_1 \rightarrow C_1 \quad b_2 \rightarrow C_2 \\
\quad \text{if } (\text{'}e\text{'} ) \quad b_1 \text{ else } b_2 \rightarrow C_e; \text{ cbr } r_e \Rightarrow l_1, l_2; l_1:\; C_1; \text{ jump } l_3; l_2:\; C_2; l_3:\ ]
Control Flow

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

while loop: `while (E) B`

```
cond:
    `rE = << E code >>`
    `cbr rE → body, done`

body:
    `<< B code >>`
    `jmp cond`

done:
```
while loop: \textbf{while (E) B}

\begin{align*}
\text{cond:} & \quad ; \text{CONTINUE target} \\
\text{rE} & = \texttt{<< E code >>} \\
\text{cbr rE} & \rightarrow \text{body, done} \\
\text{body:} & \quad \texttt{<< B code >>} \\
\text{jmp cond} & \\
\text{done:} & \quad ; \text{BREAK target}
\end{align*}

\[
\text{SWhile} \quad e \rightarrow \langle C_e, r_e \rangle \quad b \rightarrow C_b \\
\text{while `(' e ')' b} \rightarrow l_1::; C_e; \text{cbr r_e} \Rightarrow l_2,l_3; l_2::; C_b; \text{jump} l_1; l_3:
\]
for loop: \textbf{for} V \textbf{in} E1, E2 B

\begin{align*}
rX &= \text{\textless\textless{} E1 code \textgreater\textgreater{}} \\
rY &= \text{\textless\textless{} E2 code \textgreater\textgreater{}} \\
rV &= rX
\end{align*}

\textbf{cond:}
\begin{align*}
&\text{cmp\_GE} \ rV, \ rY \rightarrow rC \\
&\text{cbr} \ rC \rightarrow \text{done, body}
\end{align*}

\textbf{body:}
\begin{align*}
&\text{\textless\textless{} B code \textgreater\textgreater{}} \\
rV &= rV + 1 \quad ; \ \text{CONTINUE target} \\
&\text{jmp} \ \text{cond}
\end{align*}

\textbf{done:} \quad ; \ \text{BREAK target}
switch statement:

```java
switch (E) {
    case V1:  B1
    case V2:  B2
    default:  BD
}
```

```
rE = << E code >>
if rE == V1 goto b1
if rE == V2 goto b2
<< BD code >>
jmp end
b1:
    << B1 code >>
jmp end
b2:
    << B2 code >>
jmp end
l3:
```

NOT CURRENTLY IN DECAF
For sequential values starting with constant:

("jump table")

\[
\begin{align*}
\text{Re} &= \ll E \text{ code } \gg \\
\text{jmp } (\text{jt}+\text{Re}) \\
\text{jt: jmp l1} \\
\text{jmp l2} \\
(...)
\end{align*}
\]
• **Static single-assignment**
  - Unique name for each newly-calculated value
  - Values are collapsed at control flow points using Φ-functions
    • Useful for various types of analysis
    • Φ-functions have no actual effect at runtime
  - We’ll generate ILOC in SSA for P5
    • Unique temporary register for each newly-calculated value

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

- Procedures are harder
  - (recall x86-64 calling conventions from CS 261)
  - Need rules for control transfer, parameter passing, return values, and register usage
    - Usually specified by an application binary interface (ABI)
  - We'll cover all of this next week