Code Generation

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

Source code

```
char data[20];
int main() {
    float x = 42.0;
    return 7;
}
```

Current focus

"Back end"

Optimizations &
Machine code gen &
Assembling/linking

Checked AST + Symtables

Lexing &
Parsing &
Analysis

Code Generation

Linear IR code

✅ ✅ ✅ ✅ ✅

Checked AST + Symtables

```
main:
    loadI 7 => RET
```

Machine code

```
7f 45 4c 46 01
01 01 00 00 00
... 00 00 00 00 00
```
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
Obstacles

- 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
• Linear IR based on research compiler from Rice
• “Intermediate Language for an Optimizing Compiler”

```python
def int main()
{
    return 3+4;
}
```

```assembly
main:
    loadI 3 => r0
    loadI 4 => r1
    add r0, r1 => r2
    i2i r2 => RET
    return
```
• Simple von Neumann architecture
  - Not an actual hardware architecture, but useful for teaching
  - 64-bit words w/ 64K address space
  - Read-only code region indexed by instruction
  - Unlimited 64-bit integer virtual registers (r1, r2, …)
  - Four special-purpose registers:
    • IP: instruction pointer
    • SP: stack pointer
    • BP: base pointer
    • RET: return value
• See Appendix A (and P4 code/documentation)
• I have made some modifications to simplify P4
  - 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
<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, op2 =&gt; op3</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
</tr>
<tr>
<td>sub</td>
<td>op1, op2 =&gt; op3</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
</tr>
<tr>
<td>mult</td>
<td>op1, op2 =&gt; op3</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
</tr>
<tr>
<td>div</td>
<td>op1, op2 =&gt; op3</td>
<td>reg</td>
<td>reg</td>
<td>reg</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>reg</td>
</tr>
<tr>
<td>neg</td>
<td>op1 =&gt; op2</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
</tr>
</tbody>
</table>

<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>Boolean Arithmetic</strong></td>
</tr>
<tr>
<td>and</td>
<td>op1, op2 =&gt; op3</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
</tr>
<tr>
<td>or</td>
<td>op1, op2 =&gt; op3</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
</tr>
<tr>
<td>not</td>
<td>op1 =&gt; op2</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
</tr>
</tbody>
</table>

<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>Data Movement</strong></td>
</tr>
<tr>
<td>i2i</td>
<td>op1  =&gt; op2</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
</tr>
<tr>
<td>loadI</td>
<td>op1  =&gt; op2</td>
<td>imm</td>
<td>reg</td>
<td>reg</td>
</tr>
<tr>
<td>load</td>
<td>[op1] =&gt; op2</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
</tr>
<tr>
<td>loadAI</td>
<td>[op1+op2] =&gt; op3</td>
<td>reg</td>
<td>imm</td>
<td>reg</td>
</tr>
<tr>
<td>loadAO</td>
<td>[op1+op2] =&gt; op3</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
</tr>
<tr>
<td>store</td>
<td>op1 =&gt; [op2]</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
</tr>
<tr>
<td>storeAI</td>
<td>op1 =&gt; [op2+op3]</td>
<td>reg</td>
<td>reg</td>
<td>imm</td>
</tr>
<tr>
<td>storeAO</td>
<td>op1 =&gt; [op2+op3]</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
</tr>
<tr>
<td>push</td>
<td>op1</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
</tr>
<tr>
<td>pop</td>
<td>op1</td>
<td>reg</td>
<td></td>
<td>reg</td>
</tr>
</tbody>
</table>
### Comparison

<table>
<thead>
<tr>
<th>Operation</th>
<th>Registers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp_LT op1, op2 =&gt; op3</td>
<td>reg reg reg</td>
<td>less-than comparison</td>
</tr>
<tr>
<td>cmp_LE op1, op2 =&gt; op3</td>
<td>reg 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>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label (“op1:“)</td>
<td>control flow label</td>
</tr>
<tr>
<td>jump op1</td>
<td>unconditional branch</td>
</tr>
<tr>
<td>cbr op1 =&gt; op2, op3</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>(\varphi)-function (for SSA only)</td>
</tr>
</tbody>
</table>

**x86-64:**

- cmpq %r2, %r1
- jl L1
- jmp L2

**ILOC:**

- cmp_LT r1, r2 => rE
- cbr rE => L1, L2
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
ILOC

• Sample code:

Decaf equivalent:

```decaf
def print_int(a):
    return a
print_int(2+3*4);
```

```plaintext
loadI 2 => r1
tloadI 3 => r2
tloadI 4 => r3
tmult r2, r3 => r4
tadd r1, r4 => r5
tprint r5
```
Example

• Sample code:

```plaintext
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);
```

```
// Literal (2)
// Literal (3)
// Literal (4)
// BinaryOp (*)
// BinaryOp (+)
// FuncCall (print_int)
```

```
 FuncCall (print_int)  
   BinOp (+)  
     Lit(2)    BinOp (*)  
      Lit(3)    Lit(4)  
```

```
SInt

\[ \text{INT} \rightarrow \langle \text{loadI INT} \Rightarrow r, r \rangle \]

SAdd

\[ e_1 \rightarrow \langle C_1, r_1 \rangle \quad e_2 \rightarrow \langle C_2, r_2 \rangle \]
\[ e_1 \text{`+`} e_2 \rightarrow \langle C_1; C_2; \text{add} r_1, r_2 \Rightarrow r_3, r_3 \rangle \]

(similar for SSub (-), SMul (*), SDiv (/), SAnd (&&), and SOr (||))
2 + 3 * 4

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

Reg: r5

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

Reg: r4

Code:
- loadI 2 => r1
- Reg: 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!

<table>
<thead>
<tr>
<th>STrue</th>
<th>SFalse</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>(\text{true} \rightarrow \langle\text{loadI 1} \Rightarrow r, r\rangle)</td>
<td>(\text{false} \rightarrow \langle\text{loadI 0} \Rightarrow r, r\rangle)</td>
</tr>
</tbody>
</table>
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, 8-byte slots starting at [bp-8] (so [bp-16], [bp-24], etc.)
  - Assume we can look up base register and constant offset

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

\[
\text{loadAI } [bp-4] \Rightarrow r1
\]

\[
\text{loadAI } [bp-8] \Rightarrow r2
\]

\[
\text{add } r1, r2 \Rightarrow r3
\]

\[
\text{storeAI } r3 \Rightarrow [bp-12]
\]

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

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

\[
\begin{align*}
\text{SArrLoc} : & \quad e \rightarrow \langle C_e, r_e \rangle \quad x_s = \text{size(ID)} \quad r_b = \text{base(ID)} \\
& \quad \text{ID}[e] \rightarrow \langle C_e; \text{multI } r_e, x_s \Rightarrow r_o; \text{loadAO } [r_b + r_o] \Rightarrow r, r \rangle
\end{align*}
\]
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
  – Some of this complexity is covered in CS 430
• 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
if statement: \textbf{if} \ (E) \ \textbf{B1}

\begin{align*}
\text{rE} & = \langle \langle \ E \ \text{code} \ \rangle \rangle \\
\text{cbr} & \ \text{rE} \rightarrow \ b1, \ \text{skip}
\end{align*}

\begin{align*}
b1: & \quad \langle \langle \ \text{B1 code} \ \rangle \rangle \\
\text{skip:} &
\end{align*}

\[ \text{SIf} \quad \frac{e \rightarrow \langle C_e, r_e \rangle \quad b \rightarrow C_b}{\text{if} \ ('e') \ b \rightarrow C_e; \ \text{cbr} \ r_e \Rightarrow l_1, l_2; \ l_1;; \ C_b; \ l_2;} \]
if statement: \texttt{if (E) B1 else B2}

\[ rE = \lll E \text{ code } \rrr \]
\[ \text{cbr } rE \rightarrow b1, b2 \]

\begin{align*}
b1: & \quad \lll B1 \text{ code } \rrr \\
& \quad \text{jmp done} \\
b2: & \quad \lll B2 \text{ code } \rrr \\
\end{align*}

done:
Control Flow

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

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

\textbf{cond:}
\begin{align*}
    rE &= \textless\textless \ E \ \text{code} \ \textgreater\textgreater \\
    \text{cbr} \ rE &\rightarrow \text{body, done}
\end{align*}

\textbf{body:}
\begin{align*}
    \textless\textless \ B \ \text{code} \ \textgreater\textgreater \\
    \text{jmp} \ \text{cond}
\end{align*}

\textbf{done:}
Control Flow

while loop: while (E) B

cond: ; CONTINUE target

rE = << E code >>
cbr rE → body, done

body:

<< B code >>
jmp cond

done: ; BREAK target

\[
\text{SWhile} \quad \begin{align*}
  e &\rightarrow \langle C_e, r_e \rangle \quad b \rightarrow C_b \\
\end{align*}
\]

\[
\text{while } \langle '\ e ' \rangle \ b \rightarrow l_1; ; C_e; \ cbr \ r_e \Rightarrow \ l_2,l_3; \ l_2; ; C_b; \ jump \ l_1; \ l_3;
\]
Control Flow

for loop: for V in E1, E2 B

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

cond:
\[
\text{cmp\_GE } rV, rY \rightarrow rC
\]
\[
\text{cbr } rC \rightarrow \text{done, body}
\]

body:
\[
\text{E code}
\]
\[
 rV = rV + 1 \quad ; \text{CONTINUE target}
\]
\[
\text{jmp cond}
\]

done:
\[
\text{BREAK target}
\]
switch statement:

```
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
Control Flow

For sequential values starting with constant:
("jump table")

\[ rE = \ll E \text{ code} \llr \]

\[ \text{jmp} (jt+rE) \]

\[ \text{jt: jmp l1} \]

\[ \text{jmp l2} \]

(...)


• **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 P4
    • Unique temporary register for each newly-calculated value
    • No need for Φ-functions because we'll store to memory at every assignment

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

```plaintext
cmp_LT r1, r2 → r3
cbr r3 → l1, l2

l1:
  loadI 4 → r4
  jmp l3

l2:
  loadI 8 → r5
  jmp l3

l3:
  r6 = Φ(r4, r5)
```
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
Reading Topics

- 4.4: Ad hoc syntax-directed translation
  - General concept of AST-based translation
- 5.3: Linear IRs
- 5.4: Mapping values to names
  - Intro to static single-assignment (SSA) form
- 7.1-7.5, 7.8: Basic code generation
  - Data storage, arithmetic, booleans/conditionals, arrays
  - Control flow constructs
  - Parts needed for Decaf
- 7.6-7.7: Code gen for strings and structures
  - Not needed for Decaf