CS 432  
Fall 2022  
Mike Lam, Professor

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 & Parsing & Analysis

Checked AST + Symtables

Code Generation

Current focus

"Back end"

Linear IR code

Optimizations & Machine code gen & Assembling/linking

Machine code

char data[20];
int main() {
  float 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 (\texttt{push a, push b, multiply, pop c})
  - Three-address code \((c = a + b)\)
  - Machine code \((\texttt{movq a, %eax; addq b, %eax; movq %eax, c})\)

- 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
- “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>add</td>
<td>op1, op2</td>
<td>=&gt;</td>
<td>op3</td>
<td>addition</td>
</tr>
<tr>
<td>sub</td>
<td>op1, op2</td>
<td>=&gt;</td>
<td>op3</td>
<td>subtraction</td>
</tr>
<tr>
<td>mult</td>
<td>op1, op2</td>
<td>=&gt;</td>
<td>op3</td>
<td>multiplication</td>
</tr>
<tr>
<td>div</td>
<td>op1, op2</td>
<td>=&gt;</td>
<td>op3</td>
<td>division</td>
</tr>
<tr>
<td>addI</td>
<td>op1, op2</td>
<td>=&gt;</td>
<td>op3</td>
<td>addition w/ constant</td>
</tr>
<tr>
<td>multiI</td>
<td>op1, op2</td>
<td>=&gt;</td>
<td>op3</td>
<td>multiplication w/ constant</td>
</tr>
<tr>
<td>neg</td>
<td>op1</td>
<td>=&gt;</td>
<td>op2</td>
<td>arithmetic negation</td>
</tr>
</tbody>
</table>

**Integer Arithmetic**

<table>
<thead>
<tr>
<th>Form</th>
<th>Op1</th>
<th>Op2</th>
<th>Op3</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>op1, op2</td>
<td>=&gt;</td>
<td>op3</td>
<td>boolean AND</td>
</tr>
<tr>
<td>or</td>
<td>op1, op2</td>
<td>=&gt;</td>
<td>op3</td>
<td>boolean OR</td>
</tr>
<tr>
<td>not</td>
<td>op1</td>
<td>=&gt;</td>
<td>op2</td>
<td>boolean NOT</td>
</tr>
</tbody>
</table>

**Boolean Arithmetic**

<table>
<thead>
<tr>
<th>Form</th>
<th>Op1</th>
<th>Op2</th>
<th>Op3</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>i2i</td>
<td>op1</td>
<td>=&gt;</td>
<td>op2</td>
<td>register copy</td>
</tr>
<tr>
<td>loadI</td>
<td>op1</td>
<td>=&gt;</td>
<td>op2</td>
<td>load integer constant</td>
</tr>
<tr>
<td>load</td>
<td>[op1]</td>
<td>=&gt;</td>
<td>op2</td>
<td>load from address</td>
</tr>
<tr>
<td>loadAI</td>
<td>[op1+op2] =&gt;</td>
<td>op3</td>
<td>reg imm reg</td>
<td>load from base + immediate</td>
</tr>
<tr>
<td>loadAO</td>
<td>[op1+op2] =&gt;</td>
<td>op3</td>
<td>reg reg reg</td>
<td>load from base + offset</td>
</tr>
<tr>
<td>store</td>
<td>op1 =&gt;</td>
<td>[op2]</td>
<td>reg reg</td>
<td>store to address</td>
</tr>
<tr>
<td>storeAI</td>
<td>op1 =&gt;</td>
<td>[op2+op3]</td>
<td>reg reg imm</td>
<td>store to base + immediate</td>
</tr>
<tr>
<td>storeAO</td>
<td>op1 =&gt;</td>
<td>[op2+op3]</td>
<td>reg reg reg</td>
<td>store to base + offset</td>
</tr>
<tr>
<td>push</td>
<td>op1</td>
<td>=&gt;</td>
<td>[op2+op3]</td>
<td>push onto stack</td>
</tr>
<tr>
<td>pop</td>
<td>op1</td>
<td>=&gt;</td>
<td>reg</td>
<td>pop from stack</td>
</tr>
</tbody>
</table>

**Data Movement**
## ILOC

### Comparison

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</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>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label (“op1:”)</td>
<td>lbl</td>
<td>control flow label</td>
</tr>
<tr>
<td>jump op1</td>
<td>lbl</td>
<td>unconditional branch</td>
</tr>
<tr>
<td>cbr op1 =&gt; op2, op3</td>
<td>reg lbl lbl</td>
<td>conditional branch</td>
</tr>
<tr>
<td>call</td>
<td>fun</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>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>print</td>
<td>reg</td>
<td>print integer to standard out</td>
</tr>
<tr>
<td>print</td>
<td>str</td>
<td>print string 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 reg</td>
<td>$\Phi$-function (for SSA only)</td>
</tr>
</tbody>
</table>

### x86-64:

```plaintext
cmpq %r2, %r1
jl L1
jmp L2
```

### ILOC:

```plaintext
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

\[
\text{SBloc} \quad \frac{s_1 \rightarrow C_1 \quad s_2 \rightarrow C_2 \quad \cdots \quad s_n \rightarrow C_n}{\{s_1, s_2, \ldots, s_n\} \rightarrow C_1; \ C_2; \ \cdots \ C_n}
\]
Operational Semantics

• Expressions vs. statements
  – Former need a temporary register to store results
  – Denoted in semantics using \(<C, r> pairs
    • \(C\) = “code” attribute
    • \(r\) = “reg” attribute (temporary register)

\[
\begin{align*}
\text{SInt} & : \text{INT} \rightarrow \langle \text{loadI INT} \Rightarrow r, r \rangle \\
\text{SAdd} & : e_1 \rightarrow \langle C_1, r_1 \rangle, e_2 \rightarrow \langle C_2, r_2 \rangle \\
& \quad \rightarrow \langle C_1; C_2; \text{add } r_1, r_2 \Rightarrow r_3, r_3 \rangle \\
\text{SBlock} & : s_1 \rightarrow C_1, s_2 \rightarrow C_2, \ldots, s_n \rightarrow C_n \\
& \quad \rightarrow \langle \{s_1, s_2, \ldots, s_n\} \Rightarrow C_1; C_2; \ldots; C_n \rangle
\end{align*}
\]
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);
```
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);
  ```

- Diagram:

  ![Diagram](image)

  SInt
  
  | INT → \langle loadI INT => r, r \rangle |
  |
  | SAdd \quad e_1 → \langle C_1, r_1 \rangle \quad e_2 → \langle C_2, r_2 \rangle |
  |
  | e_1 ' + ' e_2 → \langle C_1; C_2; add r_1, r_2 => r_3, r_3 \rangle |
  (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

Code:
loadI 3 => r2
Reg: r2

Code:
loadI 4 => r3
Reg: r3
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>\langle \text{loadI 1} \Rightarrow r, r \rangle</td>
<td>\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 address into temporary base register first (zero 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

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

```
loadAI [bp-8] => r1
loadAI [bp-16] => r2
add r1, r2 => r3
storeAI r3 => [bp-24]
```

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

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

- **1-dimensional case:** \( \text{base} + \text{size} \times \text{i} \)
- **Generalization for multiple dimensions:**
  - \( \text{base} + (\text{i}_1 \times \text{n}_1) + (\text{i}_2 \times \text{n}_2) + \ldots + (\text{i}_k \times \text{n}_k) \)
- **Alternate definition:**
  - 1d: \( \text{base} + \text{size} \times (\text{i}_1) \)
  - 2d: \( \text{base} + \text{size} \times (\text{i}_1 \times \text{n}_2 + \text{i}_2) \)
  - nd: \( \text{base} + \text{size} \times (( \ldots ((\text{i}_1 \times \text{n}_2 + \text{i}_2) \times \text{n}_3 + \text{i}_3) \ldots ) \times \text{n}_k + \text{i}_k) \)
- **Row-major vs. column-major**
- **In Decaf:** row-major one-dimensional global arrays

\[
\text{SArrLoc} \quad \begin{array}{c}
\text{e} \rightarrow \langle C_e, r_e \rangle \\
\text{x}_s = \text{size}(\text{ID}) \\
\text{r}_b = \text{base}(\text{ID})
\end{array} \\
\text{ID}[e] \rightarrow \langle C_e; \text{multI} \quad r_e, x_s \Rightarrow r_o; \text{loadA0} \quad [r_b+r_o] \Rightarrow r, r \rangle
\]
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 `new label()` 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
  – Concatenate code, labels, and jumps
Control Flow

if statement: if (E) B1

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

\text{cbr rE} \rightarrow \text{b1, skip}

\text{b1:}

\ll B1 \text{ code } \gg

\text{skip:}

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

\[
\text{if } \langle 'e' \rangle \quad b \rightarrow C_e; \quad \text{cbr } r_e \Rightarrow l_1, l_2; \quad l_1::; \quad C_b; \quad l_2:\]
Control Flow

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

\begin{align*}
    rE &= << E \text{ code } >> \\
    \text{cbr } rE &\rightarrow b1, b2 \\
    b1: & \quad << B1 \text{ code } >> \\
    \quad \text{jmp done} \\
    b2: & \quad << B2 \text{ code } >> \\
\end{align*}

\textbf{SIIfElse} \begin{array}{l}
    e \rightarrow \{C_e, r_e\} \quad b_1 \rightarrow C_1 \quad b_2 \rightarrow C_2 \\
    \text{if `(` e `)`} \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:
\end{array}
Control Flow

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

cond:

\[
\text{rE} = << \text{E code} >>
\]

cbr \text{rE} → body, done

body:

\[
<< \text{B code} >>
\]

jmp cond

done:
Control Flow

while loop: `while (E) B`

cond: ; CONTINUE target

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

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

done: ; BREAK target

\[ \text{SWhile} \quad e \rightarrow \langle C_e, r_e \rangle \quad b \rightarrow C_b \]
\[ \text{while} \ ' ( \ ' e \ ' ) ^{\prime} \ b \rightarrow l_1; \quad C_e; \quad \text{cbr } r_e \rightarrow l_2, l_3; \quad C_b; \quad \text{jump } l_1; \quad l_3; \]
Control Flow

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

NOT CURRENTLY IN DECAF
switch statement:

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

```c
rE = << E code >>
if rE == V1 goto b1
if rE == V2 goto b2
<< BD code >>
jmp end
```

```c
b1:
   << B1 code >>
jmp end
```

```c
b2:
   << B2 code >>
jmp end
```

l3:

NOT CURRENTLY IN DECAF
Control Flow

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

\[ rE = << E \text{ code } >> \]
\[ \text{jmp (jt+rE)} \]
\[ \text{jt: jmp l1} \]
\[ \text{jmp l2} \]
\[ (...) \]
**Static single-assignment**

- Unique name for each newly-calculated value
- Values are collapsed at control flow points using $\Phi$-functions
  - Useful for various types of analysis
  - $\Phi$-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 $\Phi$-functions because we’ll store to memory at every assignment

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

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

l1:
    loadI 4 → r4
    jmp l3

l2:
    loadI 8 → r5
    jmp l3

l3:
    r6 = \Phi(r4, r5)
```
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
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
Allocating Symbols (pre-P4)

- Walk the AST, allocating memory for symbols
  - Each symbol has a location and offset field
    - This is a form of static coordinates
    - STATIC_VAR and static offset for global variables
    - STACK_LOCAL and BP offset for local variables
    - STACK_PARAM and BP offset for function parameters
  - Track allocated memory
    - localSize attribute for each FuncDecl
    - staticSize attribute for the Program
Walk the AST, generating code

- Build ILOC instructions for all nodes
  - Refer to operational semantics (section 7 of language reference)
  - Store in “code” attribute
  - May require copying “code” attribute of children
- Store expression results in temporary registers
  - Use “reg” attribute
  - Need state information to track the next temporary ID
  - Location loads and stores will require static coordinate info