

CS 432 Fall 2023

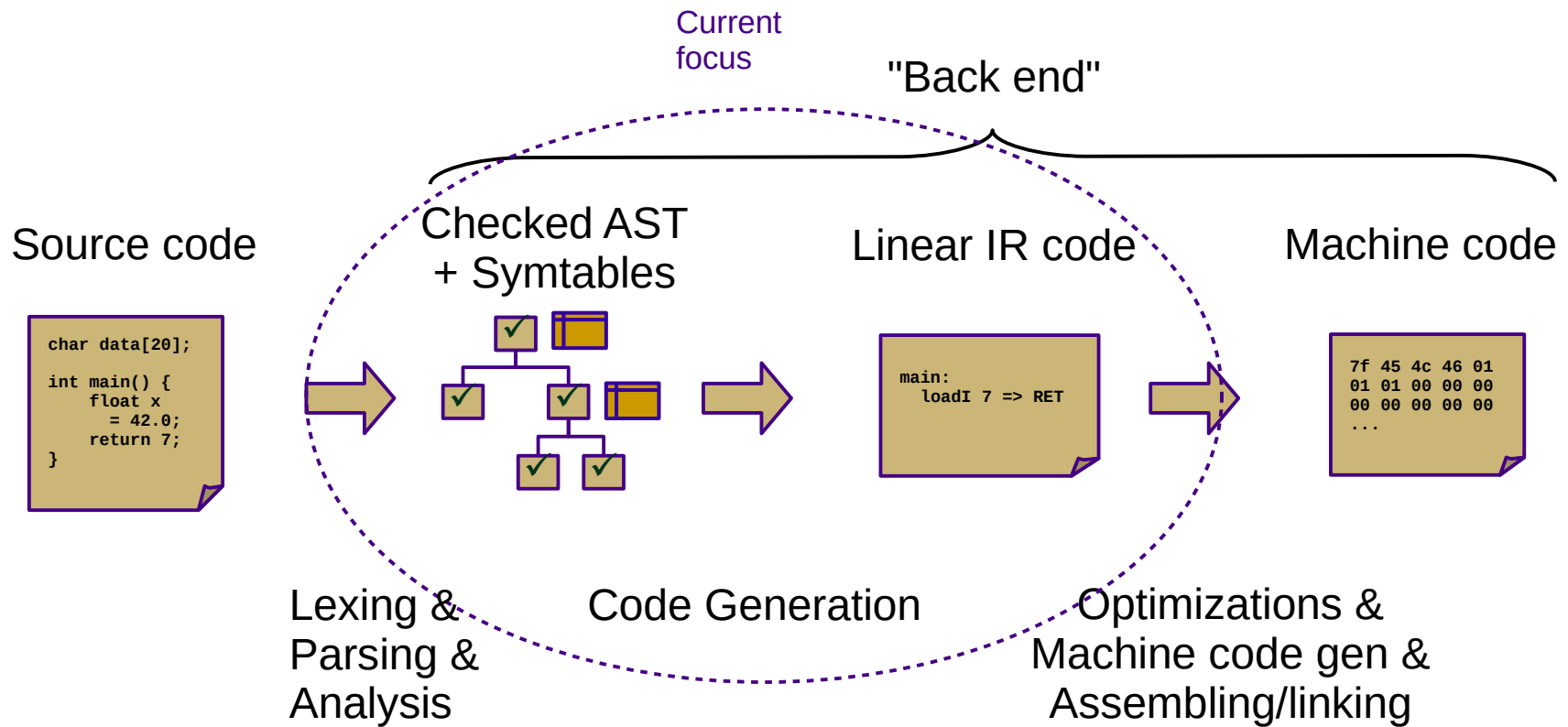
Mike Lam, Professor

```
loadI 3 => r0  
loadI 4 => r1  
mult r0, r1 => r2  
loadI 2 => r3  
add r2, r3 => r4  
print r4
```



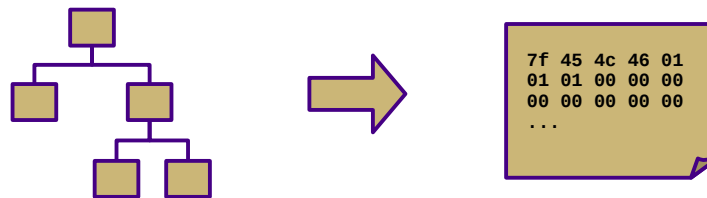
Code Generation

Compilers



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

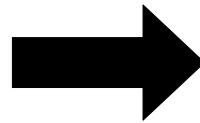
Obstacles

- 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

ILOC

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

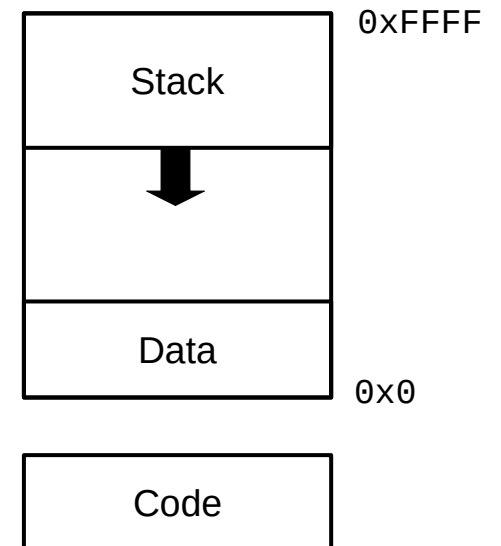
```
def int main()  
{  
    return 3+4;  
}
```



```
main:  
    loadI 3 => r0  
    loadI 4 => r1  
    add r0, r1 => r2  
    i2i r2 => RET  
return
```

ILOC

- 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 temporary virtual registers (r0, r1, r2, ...)
 - Practical limit of 2048 in reference interpreter
 - Up to 32 integer general-purpose physical registers (R0, R1, R2, ...)
 - Four special-purpose hardware registers:
 - IP: instruction pointer
 - SP: stack pointer
 - BP: base pointer
 - RET: return value



ILOC

- 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

ILOC

| Form | Op1 | Op2 | Op3 | Comment | |
|---------------------------|------------------|-----|-----|---------|----------------------------|
| Integer Arithmetic | | | | | |
| add | op1, op2 => op3 | reg | reg | reg | addition |
| sub | op1, op2 => op3 | reg | reg | reg | subtraction |
| mult | op1, op2 => op3 | reg | reg | reg | multiplication |
| div | op1, op2 => op3 | reg | reg | reg | division |
| addI | op1, op2 => op3 | reg | imm | reg | addition w/ constant |
| multiI | op1, op2 => op3 | reg | imm | reg | multiplication w/ constant |
| neg | op1 => op2 | reg | reg | | arithmetic negation |
| Boolean Arithmetic | | | | | |
| and | op1, op2 => op3 | reg | reg | reg | boolean AND |
| or | op1, op2 => op3 | reg | reg | reg | boolean OR |
| not | op1 => op2 | reg | reg | | boolean NOT |
| Data Movement | | | | | |
| i2i | op1 => op2 | reg | reg | | register copy |
| loadI | op1 => op2 | imm | reg | | load integer constant |
| load | [op1] => op2 | reg | reg | | load from address |
| loadAI | [op1+op2] => op3 | reg | imm | reg | load from base + immediate |
| loadA0 | [op1+op2] => op3 | reg | reg | reg | load from base + offset |
| store | op1 => [op2] | reg | reg | | store to address |
| storeAI | op1 => [op2+op3] | reg | reg | imm | store to base + immediate |
| storeA0 | op1 => [op2+op3] | reg | reg | reg | store to base + offset |
| push | op1 | reg | | | push onto stack |
| pop | op1 | reg | | | pop from stack |

ILOC

| Comparison | | | | |
|------------------------|-----|-----|-----|-------------------------------------|
| cmp_LT op1, op2 => op3 | reg | reg | reg | less-than comparison |
| cmp_LE op1, op2 => op3 | reg | reg | reg | less-than-or-equal-to comparison |
| cmp_EQ op1, op2 => op3 | reg | reg | reg | equality comparison |
| cmp_GE op1, op2 => op3 | reg | reg | reg | greater-than-or-equal-to comparison |
| cmp_GT op1, op2 => op3 | reg | reg | reg | greater-than comparison |
| cmp_NE op1, op2 => op3 | reg | reg | reg | inequality comparison |
| Control Flow | | | | |
| label ("op1:") | lbl | | | control flow label |
| jump op1 | lbl | | | unconditional branch |
| cbr op1 => op2, op3 | reg | lbl | lbl | conditional branch |
| call | fun | | | call function |
| return | | | | return to caller |
| Miscellaneous | | | | |
| print | reg | | | print integer to standard out |
| print | str | | | print string to standard out |
| nop | | | | no-op (do nothing) |
| phi | reg | reg | reg | φ-function (for SSA only) |

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

$$\text{SBlock} \frac{s_1 \rightarrow C_1 \quad s_2 \rightarrow C_2 \quad \dots \quad s_n \rightarrow C_n}{\text{'\{ ' } s_1, s_2, \dots, s_n \text{' } \rightarrow C_1; C_2; \dots C_n}$$

Operational Semantics

- Expressions vs. statements
 - Former need a temporary register to store results
 - Denoted in semantics using $\langle C, r \rangle$ pairs
 - C = “code” attribute
 - r = “reg” attribute (temporary register)

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

$$\text{SAdd} \frac{e_1 \rightarrow \langle C_1, r_1 \rangle \quad e_2 \rightarrow \langle C_2, r_2 \rangle}{e_1 \text{ '+' } e_2 \rightarrow \langle C_1; C_2; \text{add } r_1, r_2 \Rightarrow r_3, r_3 \rangle}$$

$$\text{SBlock} \frac{s_1 \rightarrow C_1 \quad s_2 \rightarrow C_2 \quad \dots \quad s_n \rightarrow C_n}{\text{'\{ ' } s_1, s_2, \dots, s_n \text{ '}' } \rightarrow C_1; C_2; \dots C_n}$$

Example

- Sample code:

```
loadI 2 => r0
loadI 3 => r1
loadI 4 => r2
mult r1, r2 => r3
add r0, r3 => r4
print r4
```

Decaf equivalent:

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

Example

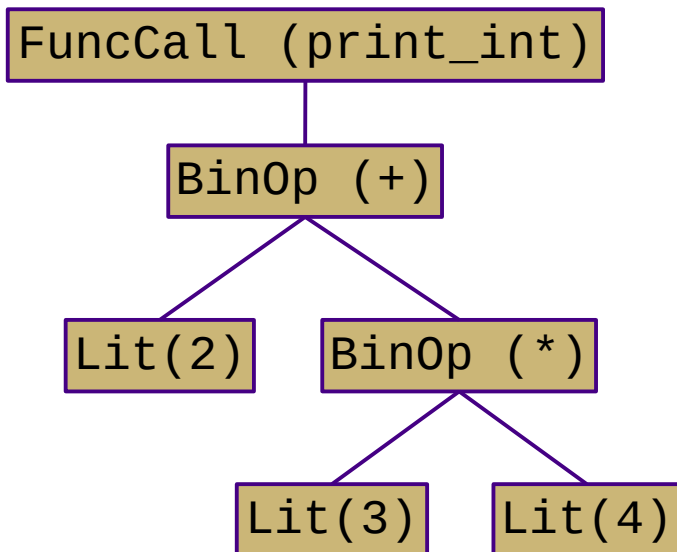
- Sample code:

```
loadI 2 => r0
loadI 3 => r1
loadI 4 => r2
mult r1, r2 => r3
add r0, r3 => r4
print r4
```

Decaf equivalent:

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

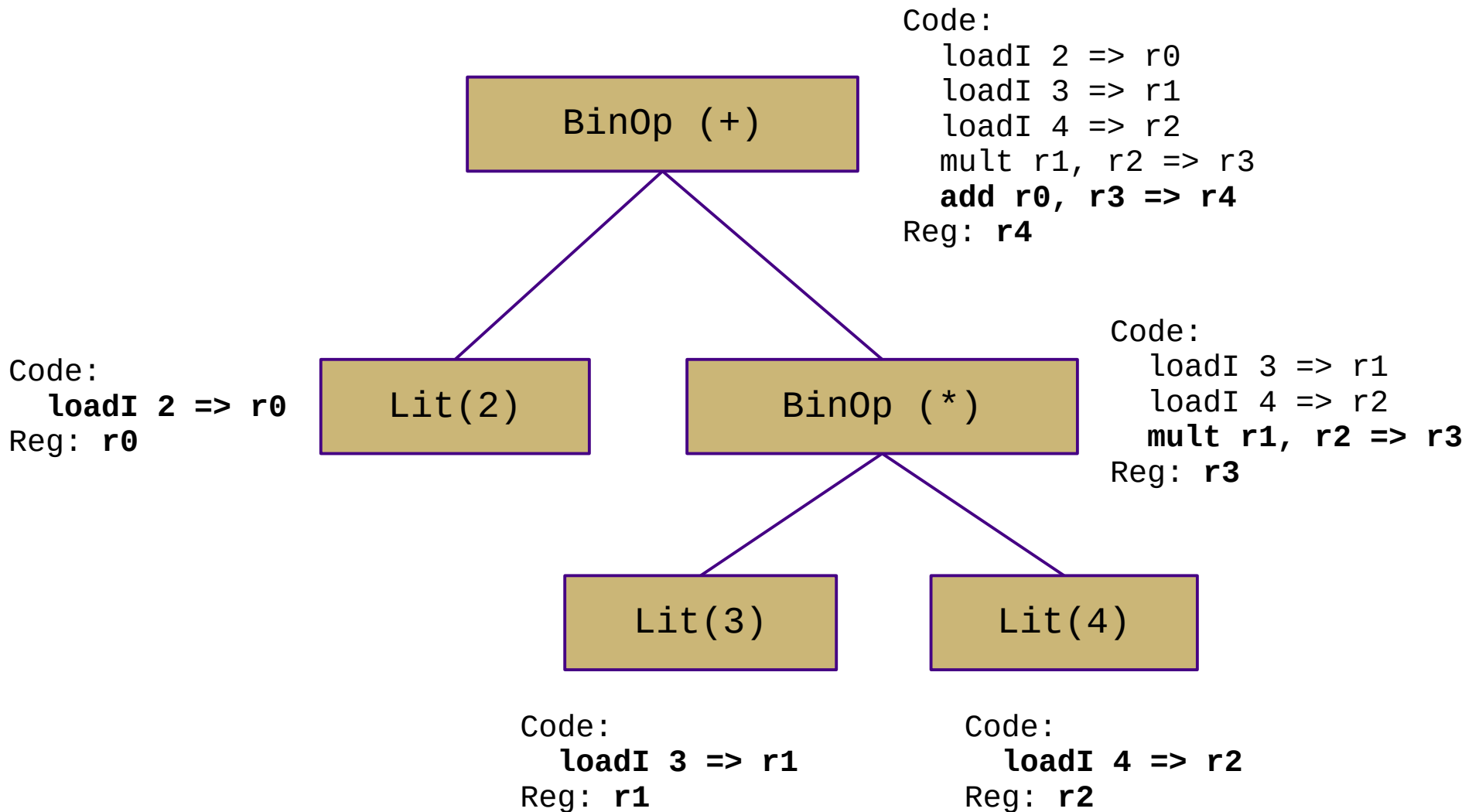
```
// Literal (2)
// Literal (3)
// Literal (4)
// BinaryOp (*)
// BinaryOp (+)
// FuncCall (print_int)
```


$$\text{SInt} \frac{}{\text{INT} \rightarrow \langle \text{loadI INT} \Rightarrow r, r \rangle}$$
$$\text{SAdd} \frac{e_1 \rightarrow \langle C_1, r_1 \rangle \quad e_2 \rightarrow \langle C_2, r_2 \rangle}{e_1 \text{ '+' } e_2 \rightarrow \langle C_1; C_2; \text{add } r_1, r_2 \Rightarrow r_3, r_3 \rangle}$$

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

Example

$2+3^*4$



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!

$S_{\text{True}} \frac{}{\text{true} \rightarrow \langle \text{loadI } 1 \Rightarrow r, r \rangle}$

$S_{\text{False}} \frac{}{\text{false} \rightarrow \langle \text{loadI } 0 \Rightarrow r, r \rangle}$

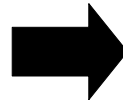
String Handling

- 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**

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



```
loadAI [bp-8] => r0  
loadAI [bp-16] => r1  
add r0, r1 => r2  
storeAI r2 => [bp-24]
```

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

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

Array Accesses

- 1-dimensional case: $\text{base} + \text{size} * i$
- Generalization for multiple dimensions:
 - $\text{base} + (i_1 * n_1) + (i_2 * n_2) + \dots + (i_k * n_k)$
- Alternate definition:
 - 1d: $\text{base} + \text{size} * (i_1)$
 - 2d: $\text{base} + \text{size} * (i_1 * n_2 + i_2)$
 - nd: $\text{base} + \text{size} * ((\dots ((i_1 * n_2 + i_2) * n_3 + i_3) \dots) * n_k + i_k)$
- **Row-major vs. column-major**
- In Decaf: row-major one-dimensional global arrays

$$\text{SArrLoc} \frac{e \rightarrow \langle C_e, r_e \rangle \quad x_s = \text{size}(\text{ID}) \quad r_b = \text{base}(\text{ID})}{\text{ID}[e] \rightarrow \langle C_e; \text{multI } r_e, x_s \Rightarrow r_o; \text{loadAO } [r_b+r_o] \Rightarrow r, r \rangle}$$

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
 - Concatenate code, labels, and jumps

Control Flow

if statement: **if (E) B1**

rE = << E code >>

cbr **rE** → b1, skip

b1:

<< **B1 code** >>

skip:

$$\text{SIf} \frac{e \rightarrow \langle C_e, r_e \rangle \quad b \rightarrow C_b}{\text{if } '(e)' \text{ } b \rightarrow C_e; \text{ cbr } r_e \Rightarrow l_1, l_2; l_1 :: C_b; l_2:}$$

Control Flow

if statement: **if (E) B1 else B2**

rE = << E code >>

cbr **rE** → b1, b2

b1:

<< **B1 code** >>

jmp done

b2:

<< **B2 code** >>

done:

$$\text{SIfElse} \frac{e \rightarrow \langle C_e, r_e \rangle \quad b_1 \rightarrow C_1 \quad b_2 \rightarrow C_2}{\text{if } \langle e \rangle \text{ } b_1 \text{ else } b_2 \rightarrow C_e; \text{ cbr } r_e \Rightarrow l_1, l_2; l_1::; C_1; \text{ jump } l_3; l_2::; C_2; l_3:}$$

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:

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*

$$\text{SWhile} \frac{e \rightarrow \langle C_e, r_e \rangle \quad b \rightarrow C_b}{\text{while } (' e ') ' b \rightarrow l_1 ;; C_e; \text{cbr } r_e \Rightarrow l_2, l_3; l_2 ;; C_b; \text{jump } l_1; l_3 :}$$

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

jmp cond

done:

**NOT CURRENTLY
IN DECAF**

; CONTINUE target

; BREAK target

Control Flow

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")

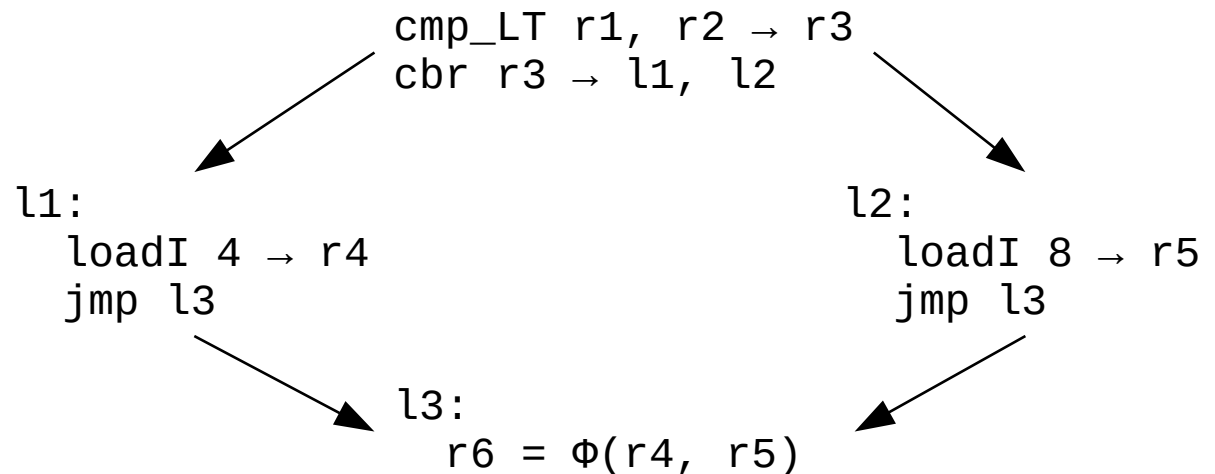
```
    rE = << E code >>  
    jmp (jt+rE)  
jt: jmp l1  
    jmp l2  
(...)
```

SSA Form

- **Static single-assignment**

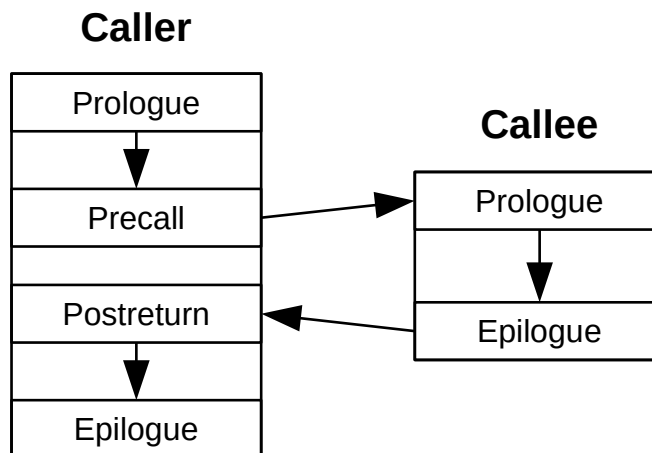
- Unique name for each newly-calculated value
- Values are collapsed at control flow points using Φ -functions
 - Useful for various types of analysis
 - Φ -functions have no actual effect at runtime
- We'll generate ILOC in SSA for P4
 - Unique temporary register for each newly-calculated value
 - No need for Φ -functions because we'll store to memory at every assignment

```
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



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

Code Generation (P4)

- 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