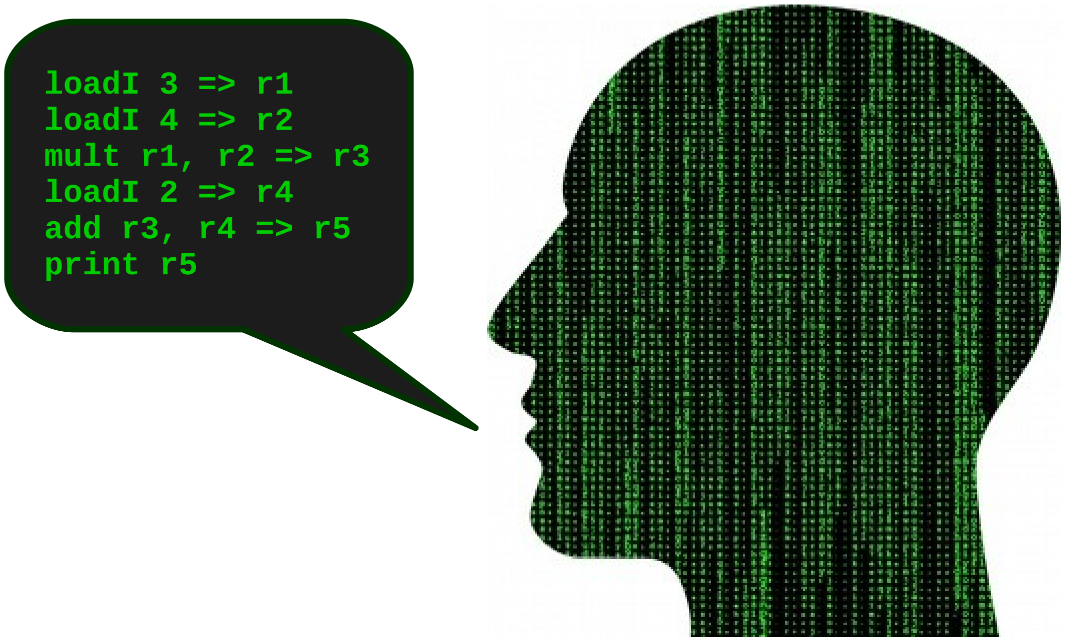
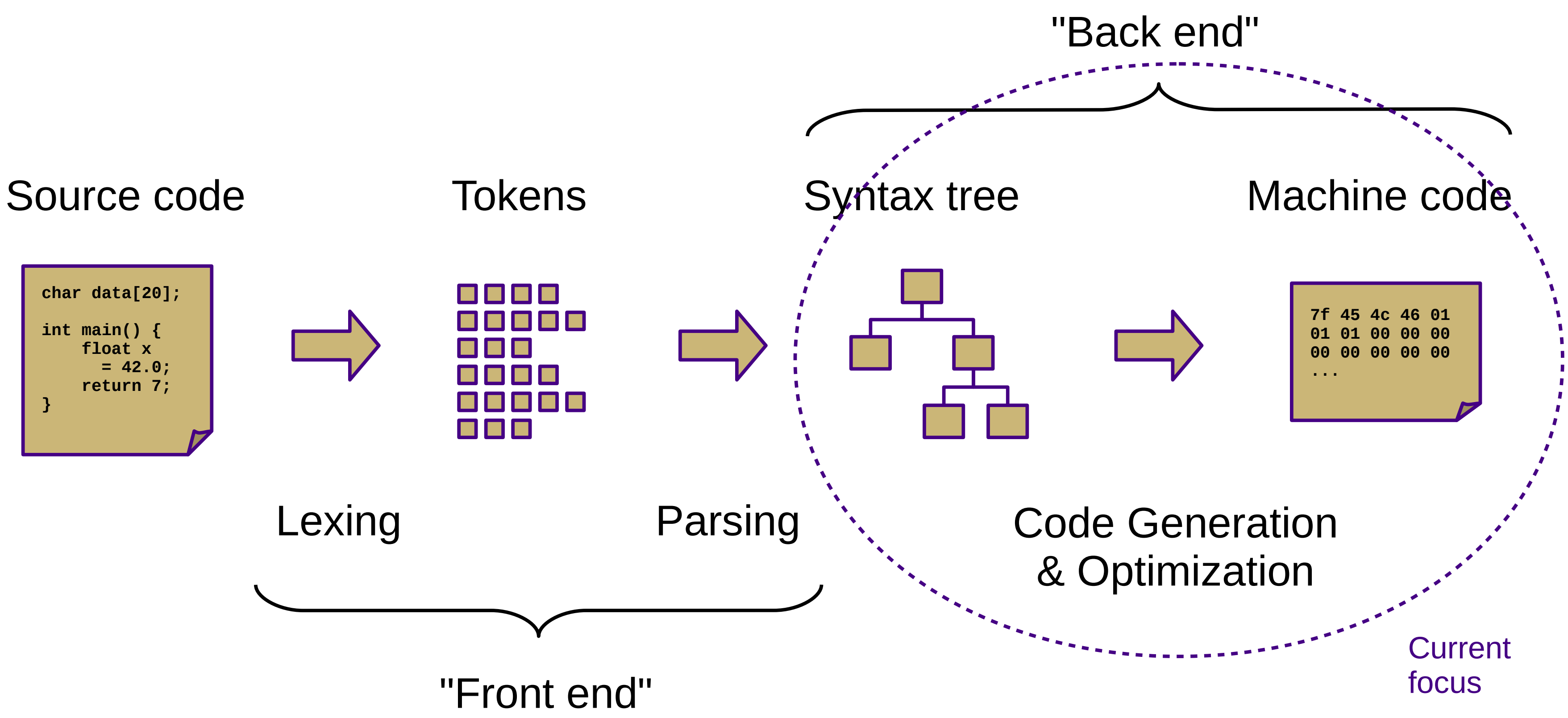
CS 432
Fall 2018

Mike Lam, Professor



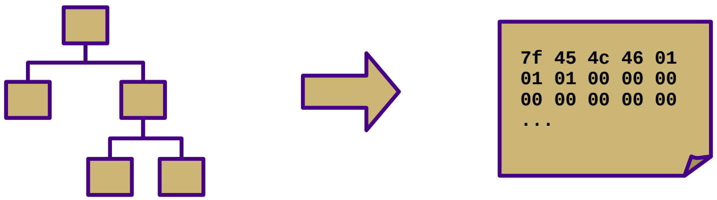
Code Generation

Compilers


"Back end"
Source code
Tokens
Syntax tree
Machine code
char data[20];
int main() {
float x
= 42.0;
return 7;
}
7f 45 4c 46 01
01 01 00 00 00
00 00 00 00 00
...
Lexing
Parsing
Code Generation
& Optimization
Current
focus
"Front end"

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 (`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

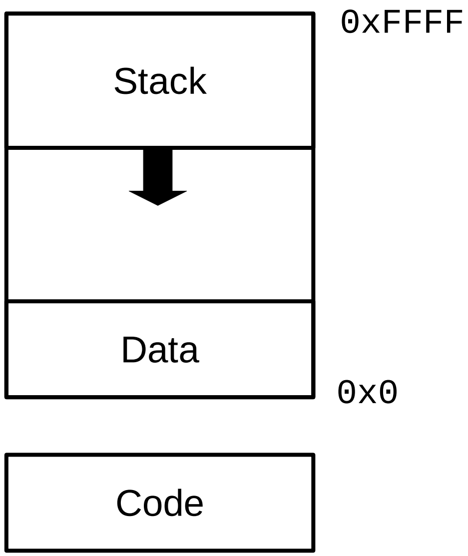
- Generating the most optimal code is undecidable
  - Unlike front-end transformations
    - (e.g., lexing & parsing)
  - Must use heuristics and approximation algorithms
  - This is why most compilers research since 1960s has been on the back end

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., `subI`)
  - Removed binary shift instructions
  - Removed character-based instructions
  - Removed jump tables
  - Removed comparison-based conditional jumps
  - Added labels and function call mechanisms (`call`, `param`, `return`)
  - Added binary `not` and arithmetic `neg`
  - Added `print` and `nop` instructions

ILOC

- 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



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
multI 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
loadS &op1 => op2	sym	reg		load symbol address
load [op1] => op2	reg	reg		load from address
loadAI [op1+op2] => op3	reg	imm	reg	load from base + immediate offset
loadAO [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 offset
storeAO op1 => [op2+op3]	reg	reg	reg	store to base + offset

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
param op1	reg			pass parameter
call	fun			call function
return				return to caller
Miscellaneous				
print	imm/ reg/ str			print value to standard out
nop				no-op (do nothing)
phi	reg	reg	reg	φ-function (for SSA only)

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”)

ILOC

- Sample code:

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
```

ILOC

- Sample code:

Decaf equivalent:

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

```
loadI 3 => r1              // ASTLiteral (3)
loadI 4 => r2              // ASTLiteral (4)
mult r1, r2 => r3          // ASTBinOp (*)
loadI 2 => r4              // ASTLiteral (2)
add r3, r4 => r5           // ASTBinOp (+)
print r5                   // ASTVoidFuncCall (print_str)
```

ILOC

- Sample code:

```
    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:

```
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: `base + width * i`
- Generalization for multiple dimensions:
  - `base + (i_1 * w_1) + (i_2 * w_2) + ... + (i_k * w_k)`
- Alternate definition:
  - 1d: `base + width * (i_1)`
  - 2d: `base + width * (i_1 * n_2 + i_2)`
  - nd: `base + width * (( ... ((i_1 * n_2 + i_2) * n_3 + i_3) ... ) * n_k + i_k) * 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

Control Flow

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

```
        rE = << E code >>
        cbr rE → b1, skip
    b1:
        << B1 code >>
    skip:
```

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:
```

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
```

Control Flow

for loop: **for V in E1, E2 B**

NOT CURRENTLY IN DECAF

```
        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
```

SSA Form

- 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

```
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 = Φ(r4, r5)
```

Procedure Calls

- These are harder
  - We'll talk about them next week