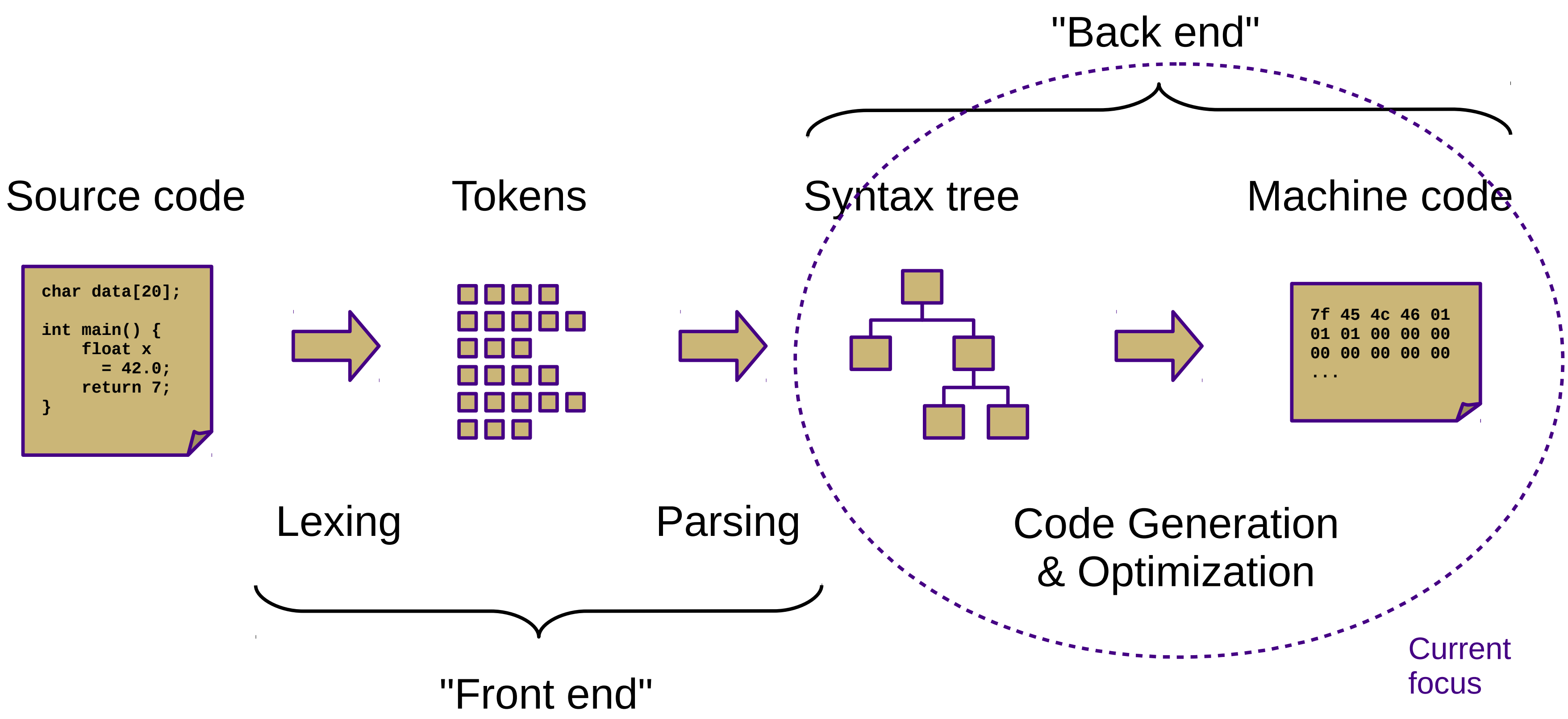
CS 480
Fall 2015

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)
    - Use a tree traversal to “linearize” the program
- But first, more general code gen topics

Goals

- Code generator outputs
  - Stack code (`push a, push b, multiply, pop c`)
  - Three-address code (`c = a + b`)
  - Machine code (`load $eax, [a]; addq $eax, [b]; store [c], $eax`)
- 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

Phases

- Instruction selection
  - Map IR to target instructions
  - Difficulty is directly related to uniformity and completeness of target instruction set
- Register allocation/assignment
  - Allocation: selecting which variables to store in registers
  - Assignment: selecting which register to use for each variable
  - General problem is NP-complete
- Instruction scheduling
  - Optimize for pipelined architectures w/ caching
  - Take advantage of speculative execution

Syntax-Directed Translation

- Similar to attribute grammars (Figure 4.15)
- Associate bits of code with each production
  - This code performs the translation or code gen
- In our project, we will use a visitor
  - Newer, cleaner technology than SDT
  - Not dependent on original grammar
- SDT is still interesting from an historical perspective
  - And useful for smaller projects

ILOC

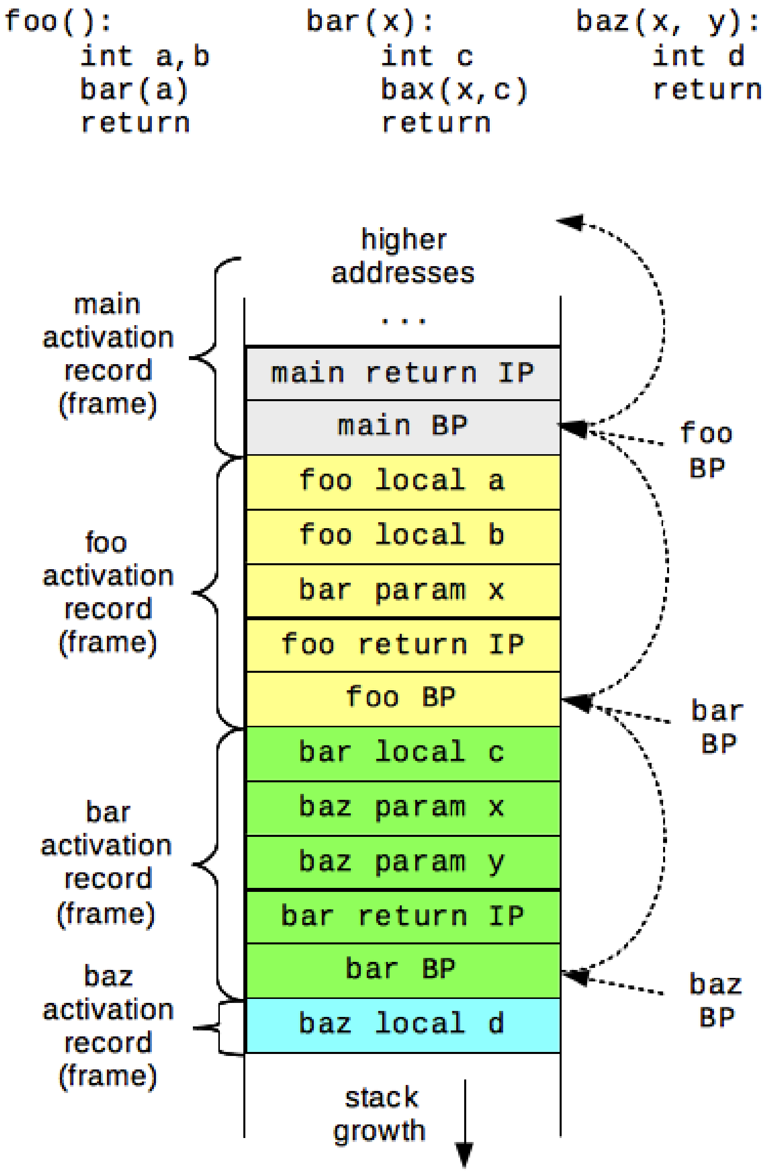
- Linear IR based on research compiler from Rice
- See Appendix A (and ILOCInstruction.java)
- I have made some modifications
  - Removed most immediate instructions (i.e., `subI`)
  - Removed binary shift instructions
  - Removed character-based instructions
  - Removed jump tables
  - Removed comparison-based conditional jumps
  - Added labels and function call mechanisms (`call`, `param`, `return`)
  - Added symbol address referencing (`loadS`)
  - Added binary `not` and arithmetic `neg`
  - Added `print` and `nop` instructions

SSA Form

- Static single-assignment
  - Naming convention that uses a 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

Assigning Storage Locations

- Memory regions
  - Code ("text")
  - Static ("data")
  - Heap
  - Stack
- Registers
  - General
  - Special



Boolean Encoding

- Integers: 0 for false, 1 for true
- Difference from book
  - No comparison-based conditional branches
- Short-circuiting
  - Not in Decaf!

Array Accesses

- Generalization to multidimensions:
  - `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

String Handling

- Arrays of chars vs. encapsulated type
  - Former is faster, latter is safer

Struct and Record Types

- How to access member values?
- OO adds a whole new level of complexity
  - Class instance records and virtual method tables

Control Flow

- Introduce program labels
  - Named location in the program
  - Generated sequentially using static newlabel() call
- Generate goto instructions using templates
  - Also called "jumps" or "branches"
  - Templates are composable

Control Flow

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

```
        << E code >>
        if E goto l1
        goto l2
    l1:
        << B1 code >>
    l2:
```

Control Flow

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

```
        << E code >>
        if E goto l1
        goto l2
    l1:
        << B1 code >>
        goto l3
    l2:
        << B2 code >>
    l3:
```

Control Flow

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

```
l1:                                     ; CONTINUE target
    << E code >>
    if E goto l2
    goto l3
l2:
    << B code >>
    goto l1
l3:                                     ; BREAK target
```

Control Flow

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

NOT CURRENTLY IN DECAF

```
        << E1 code >>
        << E2 code >>
        V = E1
    l1:
        t1 = V >= E2
        if t1 goto l2
        << B code >>
        V = V + 1                  ; CONTINUE target
        goto l1
    l2:                            ; BREAK target
```

Control Flow

switch statement:

switch (E) {

case V1: B1

case V2: B2

default: BD

}

```
        << E code >>
        if E == V1 goto l1
        if E == V2 goto l2
        << BD code >>
        goto l3
    l1:
        << B1 code >>
        goto l3
    l2:
        << B2 code >>
        goto l3
    l3:
```

NOT CURRENTLY IN DECAF

Control Flow

For sequential values starting with constant (C):

("jump table")

```
      << E code >>
      goto jt(E-C)
  jt: goto l1
      goto l2
  (...)
```

(can also use raw instruction addresses and pointer arithmetic)

Procedure Calls

- These are hard!
  - We'll talk about them next week