CIS (4|5)61

Introduction to Compiler Construction

University of Oregon

Fall 2018

"Brutal. Savage. Soul-crushing. I LOVED IT."

(Actual description of a running course by a friend of an ultra-runner friend, but it could be a description of 461/561.)

Why study compilers?

After all, few developers build compilers for general purpose languages ... BUT ...

Many (good) developers build translators

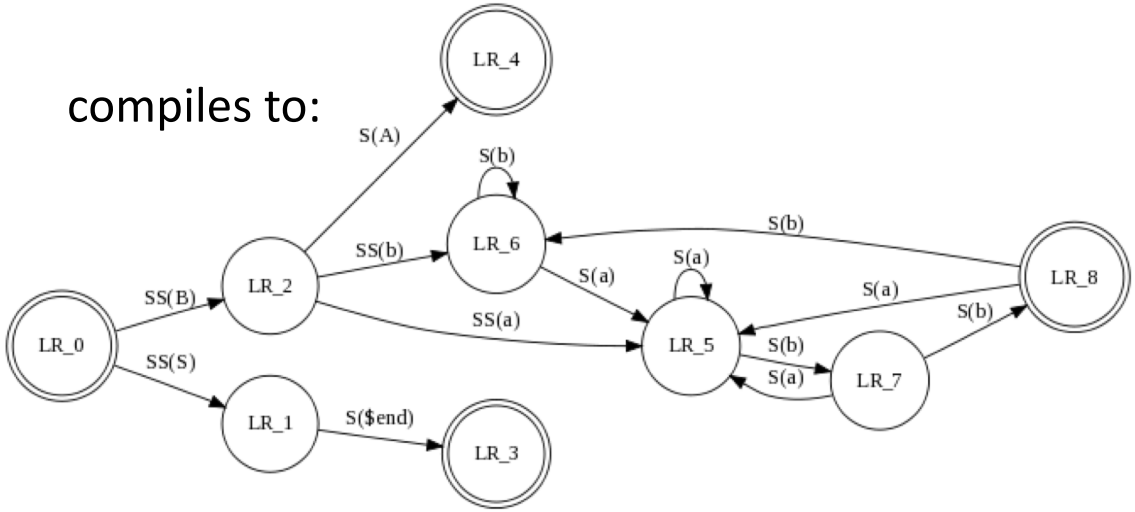
A key tool in the solutions toolbox

And besides, compiler construction is

- A case study of successful interplay between CS theory and systems
- A case study in the development of a standard application software architecture

```
digraph finite_state_machine {
        rankdir=LR;
        size="8,5"
        node [shape = doublecircle]; LR_0 LR_3 LR_4 LR_8;
        node [shape = circle];
        LR_0 -> LR_2 [ label = "SS(B)" ];
        LR_0 -> LR_1 [ label = "SS(S)" ];
        LR_1 -> LR_3 [ label = "S($end)" ];
        LR_2 -> LR_6 [ label = "SS(b)" ];
        LR_2 -> LR_5 [ label = "SS(a)" ];
        LR_2 -> LR_4 [ label = "S(A)" ];
        LR_5 -> LR_7 [ label = "S(b)" ];
        LR_5 -> LR_5 [ label = "S(a)" ];
        LR_6 -> LR_6 [ label = "S(b)" ];
        LR_6 -> LR_5 [ label = "S(a)" ];
        LR_7 -> LR_8 [ label = "S(b)" ];
        LR_7 -> LR_5 [ label = "S(a)" ];
        LR_8 -> LR_6 [ label = "S(b)" ];
        LR_8 -> LR_5 [ label = "S(a)" ];
}
```

compiles to:



Two Myths of CS Research

The pipeline myth:

First there was theory, and it was good
and then lesser creatures learned to apply it

The vacuum myth:

Systems researchers developed XXX on their own
while theoreticians wrote irrelevant papers

In compiler construction, theory *followed* practice but *utterly transformed* it

Language Paleontology

The history of compiler techniques is readable in the bones of old languages

Fortran IV, BASIC: Lexical analysis mixed with parsing. Algol 60 separates them.

- DO 20 I = 30 . 5 is an assignment
- DO 20 I = 30 , 5 is a loop head

K&R C: Primitive register allocation, programmer hints ("register" declarations) necessary for decent code;

Java: Garbage collection finally reaches the big time, after 30 years of experience and improvement

- But it ***still*** doesn't have generators, ~~closures~~, properties, ... so we write verbose, confusing code to patch around the holes.

(added in Java 8)

Recently: Type inference ("auto" in C++), optional (Swift), ...

Course Goals

- Understand (a bit of) the relevant theory
  - Lexical analysis, parsing, attribution & analysis, code improvement and generation
- Understand how compilers work
  - Standard architecture & basic techniques, including run-time organization
- Learn to use compiler construction tools
  - At least lexer & parser generators
- Understand some language design trade-offs

In 10 weeks? How?

Tight schedule for theory + project, but a complete "end-to-end" project is essential

We can cope with:

- Heavy use of tools for parsing & code generation
- Light touch on theory
- Immediate start (now!)
- Optional: 2-3 person teams in second half (static semantics & code generation phases)

Project Stages

(1) Lexical analysis (using re-Flex, JFlex, ...)

(2) Parser (using Bison, CUP, ...)

(3a) Abstract syntax & symbol table

(3b) Static semantics (type checker)

(4) Code generation (in C or LLVM)

Want to use different tools or a different implementation language? OK, but you're on your own to solve problems you encounter. I prefer LR (LALR(1), LR(k), etc) to LL(k) or recursive descent parsing, but will be flexible if your implementation language lacks an LALR(1) parser generator.

Materials

Textbook: You have options

Dragon book is good, but expensive

Several good texts ... used is ok

Implementation language: You have options

but it's up to you to help me automate testing

and you must find the tools!

Language to compile: Quack

"It's my own invention" (bad idea, probably)

Piazza: Use it!

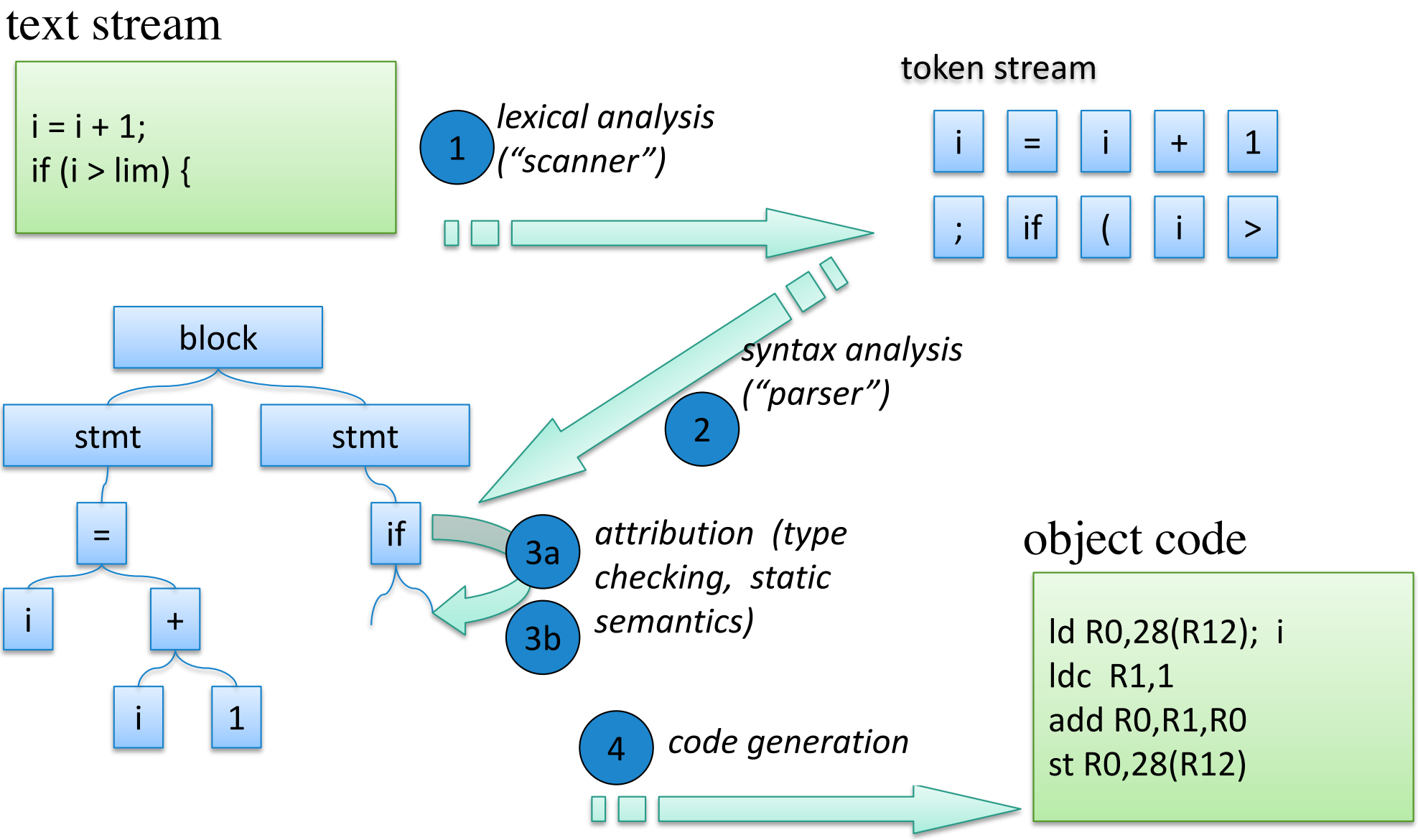
About Quack

Reference manual later this week

(I'm still trimming some rough bits)

Simple object-oriented language, single inheritance, *with type inference*

Standard Compiler Architecture



Ground Rules

Collaboration on projects

- Within team: Unlimited, but documented
- Between teams:
  - Discussion is ok, code sharing is not
  - If you aren't sure where the line is, ask me

Project Grading: Mainly by test cases

- Cases you provide: show me what works
- Cases we provide in advance
- Additional cases we make up

Turning in project

Wide latitude in language choice —

use something you are comfortable with, or that is relevant to your research project

Limited choice in packaging and turn-in

I will need to automate project testing

Delivery platforms:

- lx, with a standard shell script
- HPC platform? If you prepare it
- ??? (not too many … but I can manage one more)
  - I do not want to install your Linux image in VirtualBox!

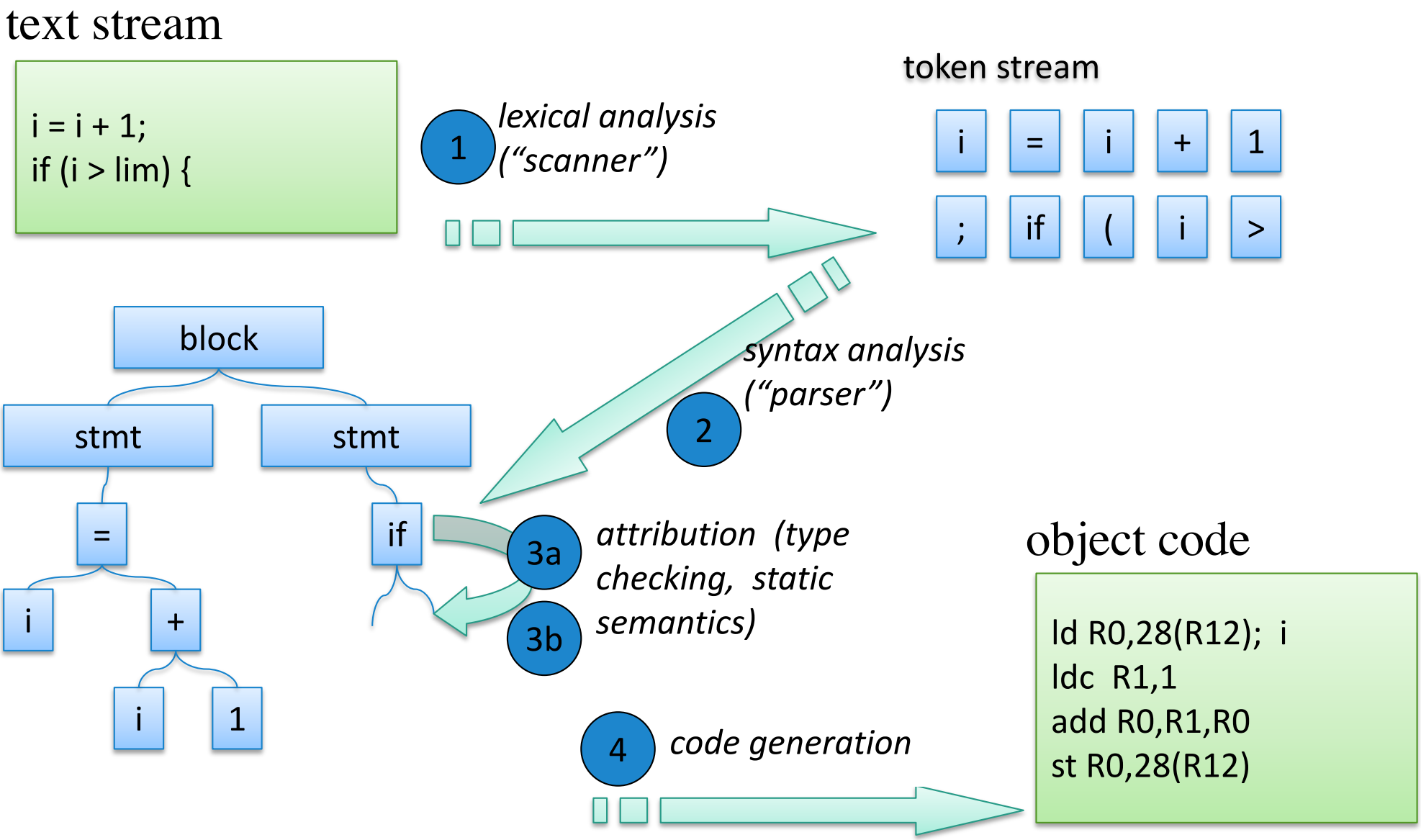
Cheating Policy

- Undocumented borrowing of ideas or code is plagiarism
  - Your best defense is to ask in advance and to carefully document what is taken from elsewhere.
  - A *lot* of borrowing is possible *with proper credit*
- Cheating will be reported and punished
  - Failure, or lower grade, or ...
  - Following UO student conduct code guidelines

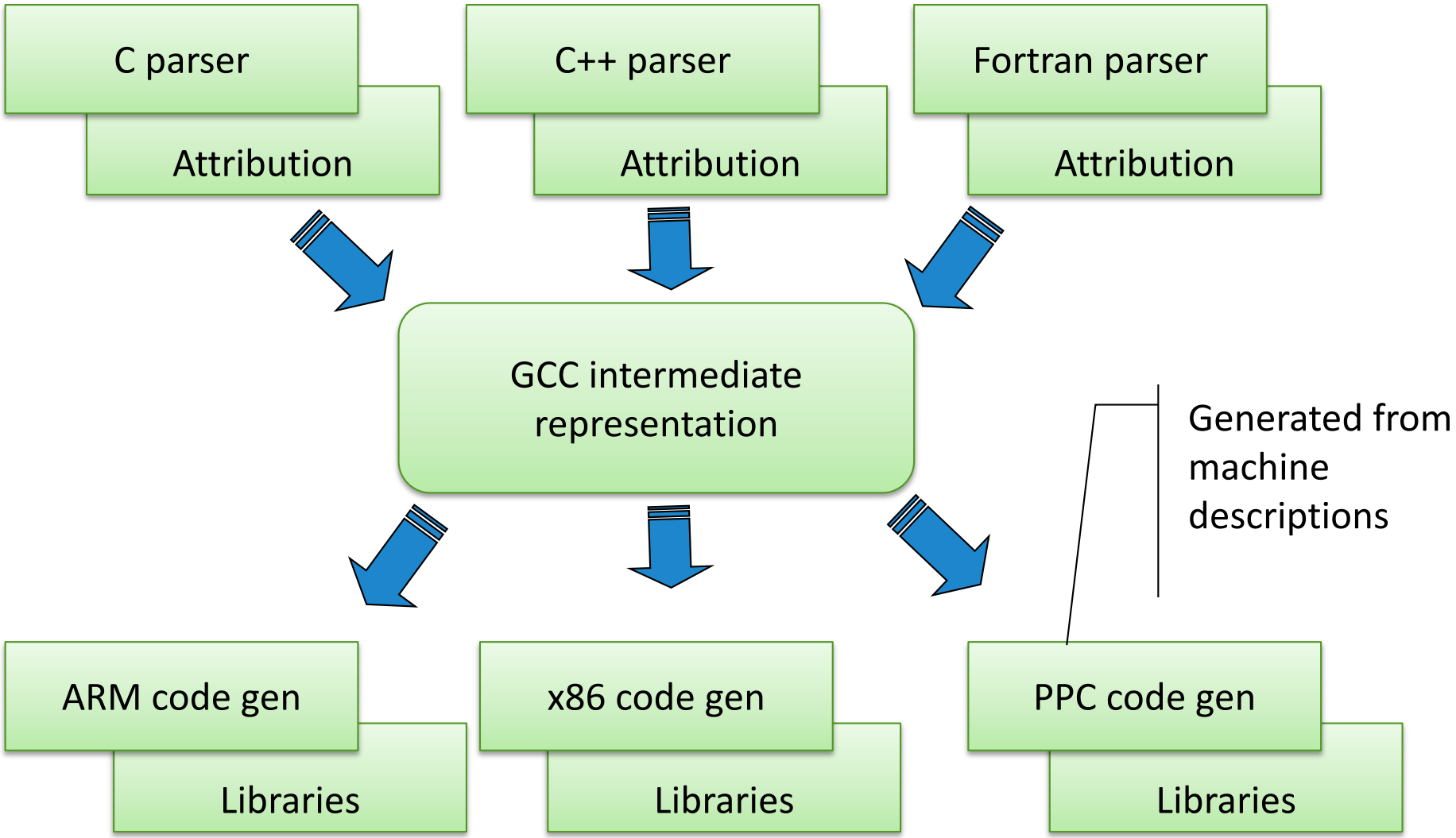
Course Grading

- 50% project total, in phases [+/- 10%]
  - with special weight on final phase: How much of the language is correctly implemented
- 50% two exams [+/- 10%]
- 461 versus 561
  - *All* students must be ready for graduate level material and a demanding project
  - Grads may have different or extra exam questions
  - Grads may choose a short paper or an extension or improvement to our project (talk to me)
- No fixed distribution of grades

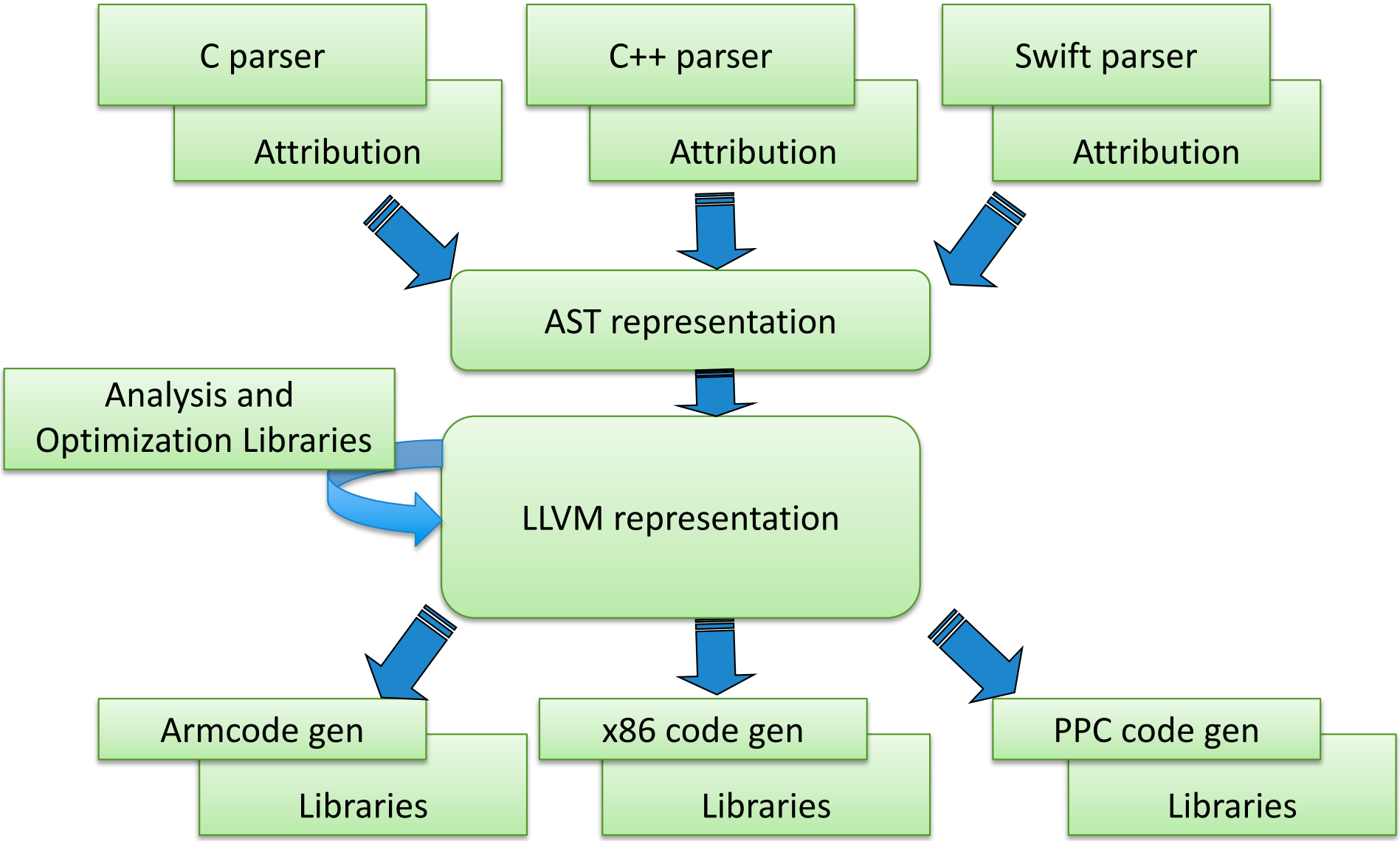
Standard Compiler Architecture



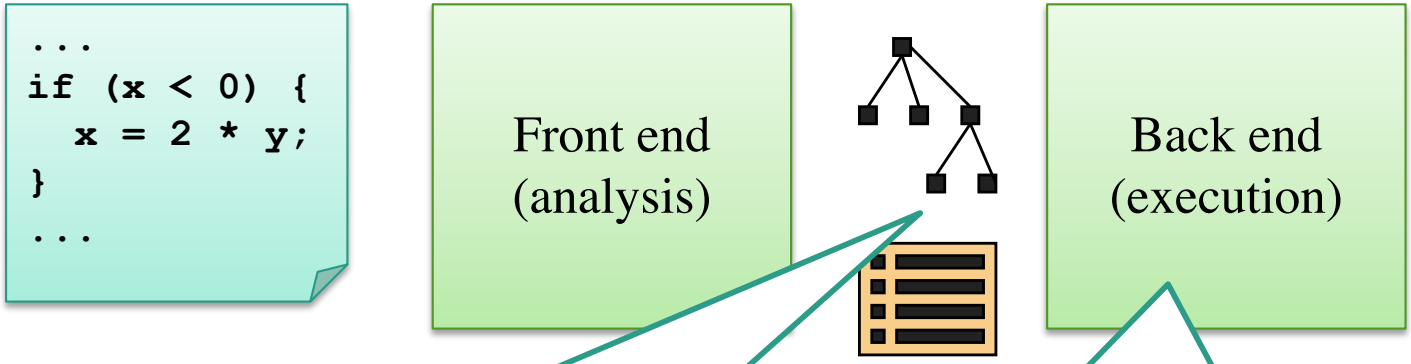
Fronts & Backs: Retargeting: gcc example



Fronts & Backs: Retargeting: Clang/LLVM



Interpreter Structure (typical)



Intermediate structure could be: trees (Lisp), byte code (Java), token stream (BASIC), ...

Engine can be a separate program(Java) or integrated (Perl, Basic, PostScript, Python, ...)

Aside: Where the research action is

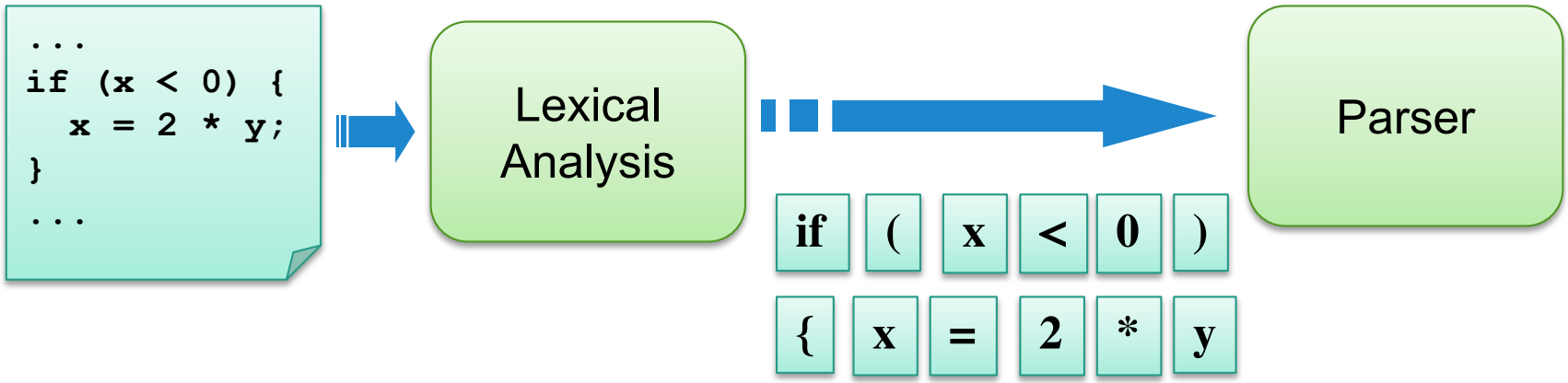
Until recently,

- Code generation for compiled languages was much hotter than anything in the front ends, although type systems were hot in theory. Interpreters were a mostly dead area.

Recently,

- Interpreters for dynamically typed languages are hot (because of browsers)
- Program analysis is as hot as optimization (because of security)
- Languages are bubbling again ... Option types? Better concurrency models? Functional features in mainstream languages?

Lexing & Parsing



Lexical analyzer reads text, produces token stream

- Theory base: regular languages, finite automata

Parser analyzes grammatical structure

- Theory base: context-free languages, LL or LR parsing

Analogy to natural (human) language

- Lexical structure:
  - Group phonemes into lexemes and words
- Syntactic structure:
  - Use grammar to understand the structure of a sentence
- Semantics:
  - Understand the meaning of a sentence
- Pragmatics and discourse structure:
  - (maybe not)

Lexical structure

The “lexemes” or “tokens” of a language are the smallest units

Examples (in Java):

( ) if . + *

myLongVariableName

“this whole String is just one token”

42

0x77ff88L

// but a comment is not a token; it is discarded

Scanning for tokens

Describe lexical structure by regular languages (regexp patterns)

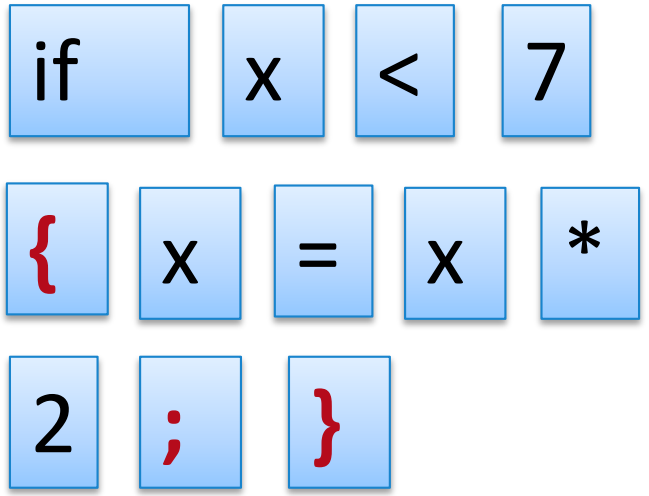
But compile into one big automaton

so we get linear time matching, independent of the number of different patterns

Tools: lex, flex, re-flex, jflex, ... (and parts of JavaCC, Antlr, ...)

Aside: Tokens that aren't there

```
if x < 7:
   x = x * 2
```



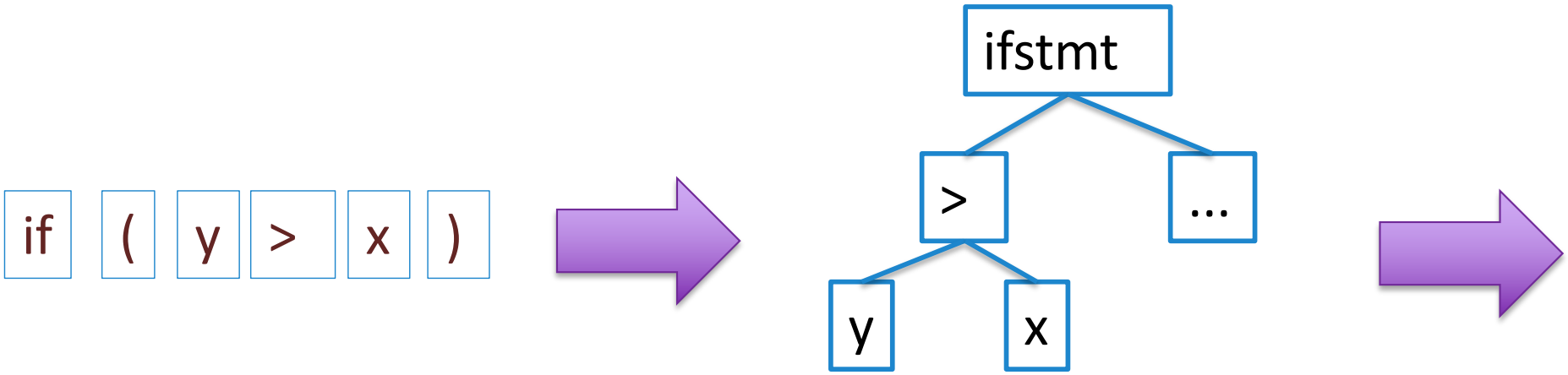
In some languages, the scanner interprets indentation or other patterns and synthesizes some tokens.

Example: Python

~~Example: Our language Quack~~

I tried. Surprisingly tricky; not appropriate for an intro class

Parsing syntactic structure



The parser uses grammar rules to find the structure of the program

stmt ::= ifstmt

ifstmt ::= if (expr) stmt else stmt

expr ::= expr relop expr

relop ::= < | > | >= | <=

Historical Note: Parsing

Modern organization of lexer, parser depends on language design

- Requires division of language into lexical and syntactic structure, as pioneered in Algol
- FORTRAN, BASIC defy lexical analysis; C requires hacks for context-sensitive lexical analysis

Theory → Parsing tools → Language design

Modern languages are designed to be parsed!

What about macros?

~~Two~~ Three flavors:

- Text macros (C)
  - String processing, not really part of the language
- Procedural token macros (Rust)
  - A little safer; powerful enough for some domain-specific notations. ”Hygienic”
- Syntax macros (Lisp)
  - Full procedural processing on abstract syntax (requires AST as first-class entity)

"Static semantic" analysis

- Type checking, overload analysis, attribution
  - Syntactic analyses that are not (efficiently) expressible in a context-free grammar
- Theory: Type systems (inference rules), attribute grammars, data flow analysis
  - Tools exist but are little used. Most implementations are hand-coded and loosely follow theory (as will yours).
  - Language design can help: Small, regular ASTs with uniform type rules

Core Language + Syntactic Sugar

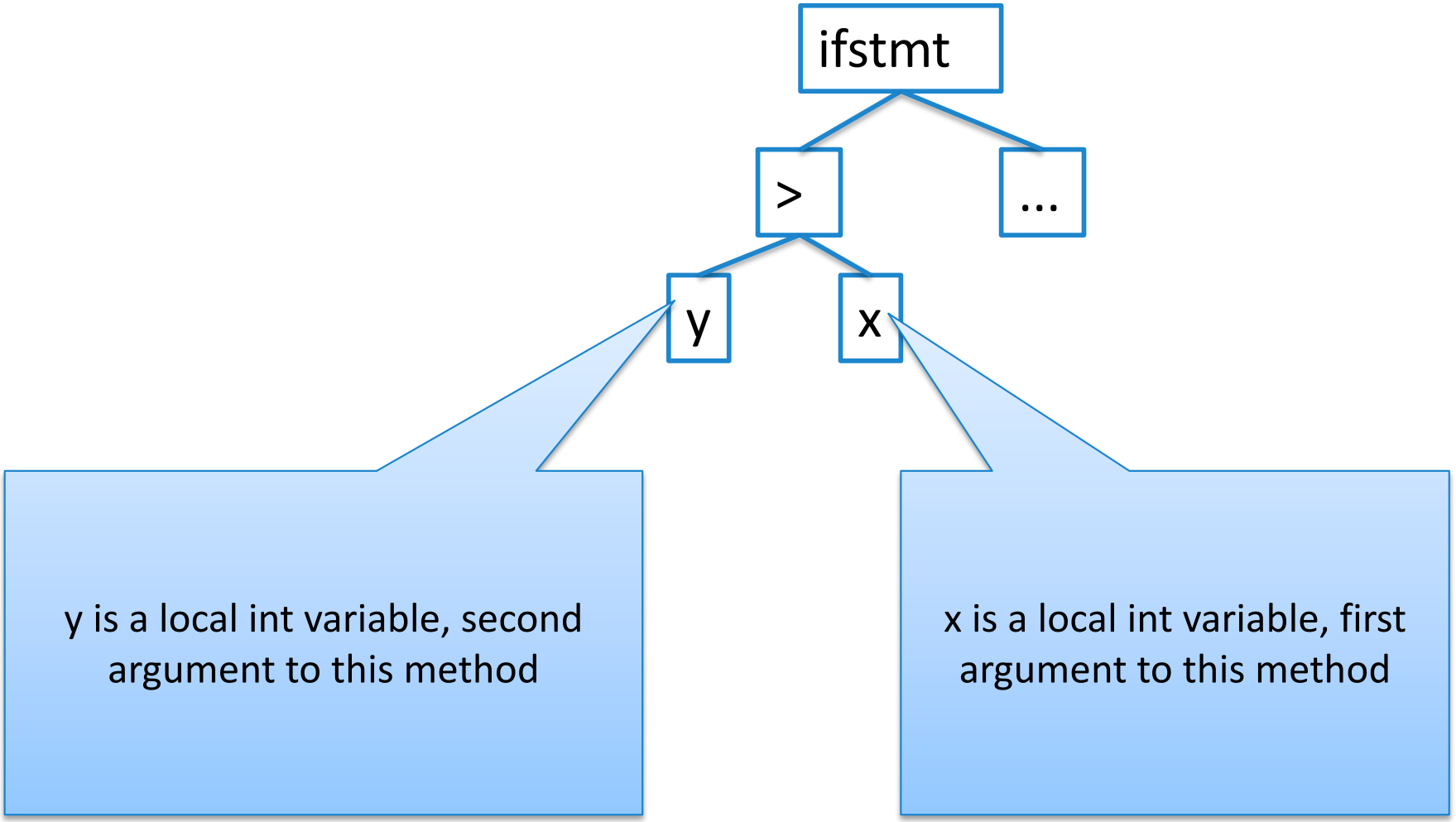
We can have a rich surface language with a simple, uniform core language by treating some syntax as *sugar*

Example: In Quack, x + y * z is sugar for

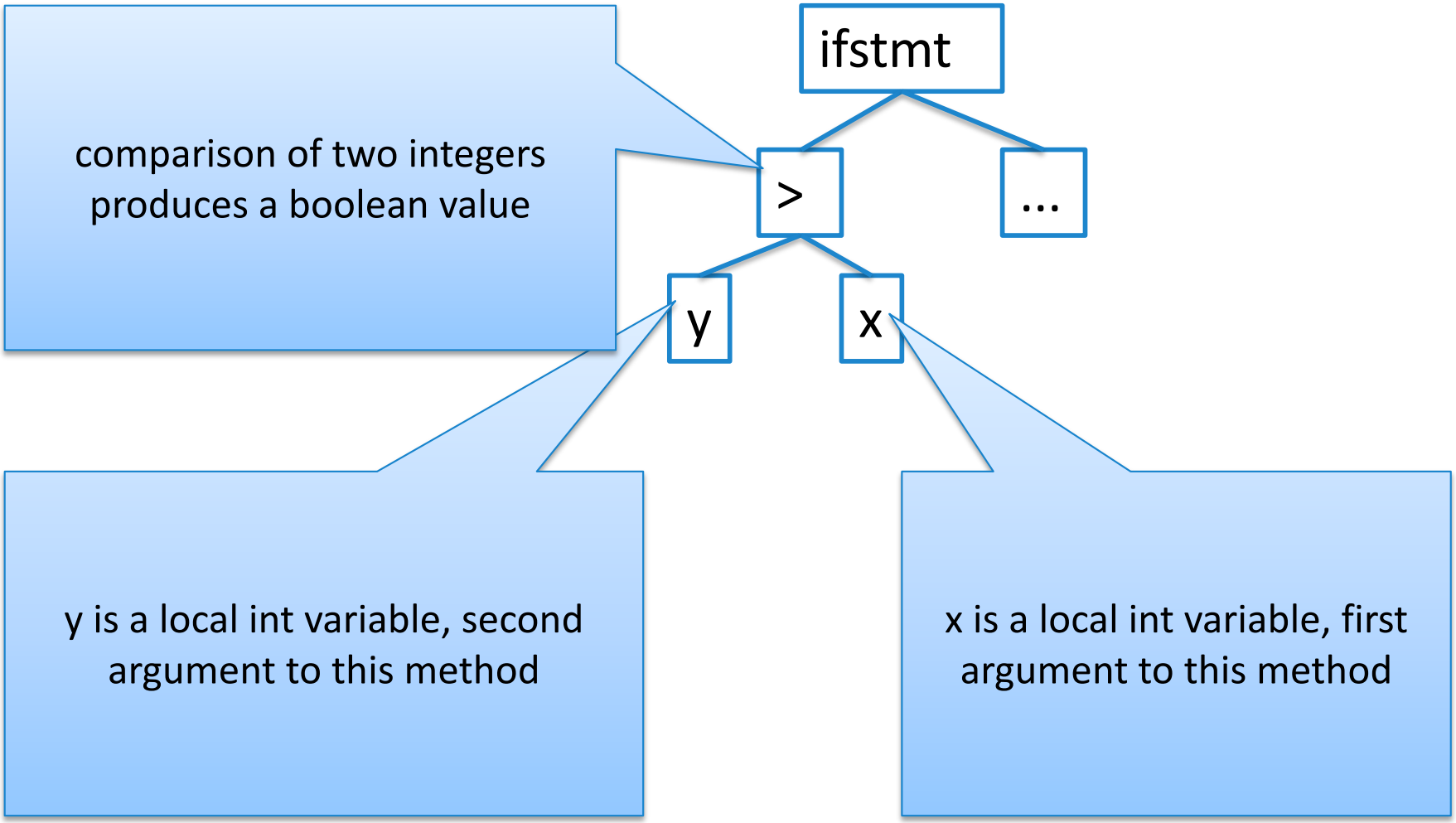
x.PLUS(y.TIMES(z))

There are no AST node types for multiplication, addition, etc; no extra rules for type checking!

"Static" semantics (scope, types, ...)



"Static" semantics (scope, types, ...)

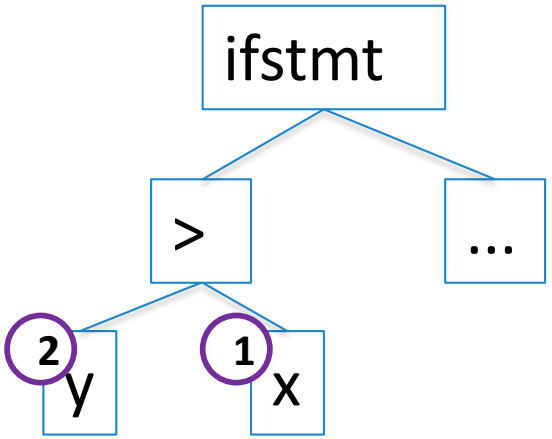


Back End Tasks (Translation)

- Assignment of run-time structures
  - Frame layout, register allocation
  - Intimately tied to libraries (e.g., garbage collector)
- Instruction choice and emission

- Theory: Many bits and pieces
  - Register allocation by graph coloring
  - Tree-rewriting for code generation

Example: Generate Java stack code by "walking the tree"




```
0:      iload_1
1:      istore_3
2:          iload_2
3:          iload_1
4:          if_icmple   9
7:      iload_2
8:      istore_3
9:      iload_3
10:     ireturn
```

A riddle …

In Quack, we will use syntactic sugar to keep the abstract syntax (internal tree representation) as simple as possible

a * b is just shorthand for a.MULT(b), so there is no special AST node for multiplication

but there will be special nodes for *and*, *or*, *not*.

Why?

(Hint: Think about generating code)

Summary: Theory and Tools

- Front end: stable theory, robust tools
  - Lexical analyzer generators (regular languages)
  - Parser generators (context-free languages)
- Middle and back end: maturing theory, tools in flux
  - Attribute grammars
  - Data flow analysis / abstract interpretation
  - Register allocation by graph coloring
  - Code generation by rewriting (tree grammars)
- Run-time structures
  - Lots of knowledge, but no theory-supported tools

Get Started

- Download and install a scanner generator
  - RE-Flex for C/C++, Jflex for Java, or an equivalent tool if you are using a different programming language
- Study a couple of example (J)Flex programs
- Write something very simple
  - Warm up project (zero points): Scanner reads plain text with “ and ‘, substitutes HTML entities &lquo; &rquo; “ ” ' " as appropriate
- Scanner project due next Thursday
  - Scanner for the Quack language from the handout on Piazza, with a driver that just prints each token.