

CSE 401/M501 – Compilers

Overview and Administrivia

Hal Perkins

Spring 2023

Agenda

- Introductions
- Administrivia
- What's a compiler?
- Why you want to take this course 😊

Welcome back!

- This has been a strange world for the last few years and there's still a lot of stress for many
- Please speak up if things aren't (or are!) going well
 - We can often help if we know about things, so stay in touch with TAs, instructor, advising, friends and peers, others
 - Don't try to "tough it out" or pretend it will get better if you just ignore it – speak up!
- We're all in this together but not all in the same way, so please show understanding and compassion for each other and help when you can – both in and outside of class
- Stay healthy! If you do come down with something, please stay home until recovered and not contagious
 - Lectures are on panopto if you do need to miss (but this is not a remote-learning class)
- Please be realistic about your workload – it's up to you to be sure you have the time and energy to handle your academic and other commitments

Who: Course staff

- Instructor: Hal Perkins: UW faculty for a while; CSE 401 veteran (+ other compiler courses)
- TAs: Robert Burris, Rachel Phuong, John Taggart, and Randy Zhang
- Get to know us – we're here to help you succeed!
- Office hours will start in the few days – posted on calendars and announced on ed when ready. Will be a mix of in-person, hybrid, and zoom

Credits

- Some direct ancestors of this course:
 - UW CSE 401 (Chambers, Snyder, Notkin, Perkins, Ringenborg, Henry, ...)
 - UW CSE PMP 582/501 (Perkins)
 - Rice CS 412 (Cooper, Kennedy, Torczon)
 - Cornell CS 412-3 (Teitelbaum, Perkins)
 - Many books (Appel; Cooper/Torczon; Aho, [[Lam,] Sethi,] Ullman [Dragon Book]; Fischer, [Cytron ,] LeBlanc; Muchnick, ...)
- Won't attempt to attribute everything – and some (many?) of the details are lost in the haze of time

CSE M 501

- Enhanced version for 5th-year BS/MS students.
- M501 students will have to do a significant addition to the project, or some other extra work if agreed with instructor (papers, reports, ???)
 - More details later
- Otherwise 401 and M501 are the same (lectures, sections, assignments, infrastructure, ...)

So whadda ya know?

- Official prerequisites:
 - CSE 332 (data abstractions)
 - and therefore CSE 311 (Foundations)
 - CSE 351 (hardware/software interface, x86_64)
- Also very useful, but not required:
 - CSE 331 (software design & implementation)
 - CSE 341 (programming languages)
 - Who's taken these?

Lectures & Sections

- Both required
- All material posted, but they are visual aids
 - Be here! Take notes!
 - Panopto lecture recordings intended for review and unavoidable absences only
- Sections: additional examples and exercises plus project details and tools
 - We will have sections this week

Gadgets in class

- Gadgets reduce focus and learning
 - Bursts of info (*e.g.* notifications, IMs, etc.) are *addictive*
 - Heavy multitaskers have more trouble focusing and shutting out irrelevant information
 - <http://www.npr.org/2016/04/17/474525392/attention-students-put-your-laptops-away>
- So how should we deal with laptops/phones/etc.?
 - Just say no!
 - No open gadgets during class (really!)
 - Unless you are actually using a device to take notes or for other appropriate uses....
 - Urge to search? – ask a question! Everyone benefits!!
 - You may close/turn off non-notetaking electronics now
 - Pull out a piece of paper and pen/pencil instead 😊

Communications

- Course web site (www.cs.uw.edu/401)
- Discussion board – ed
 - For anything related to the course
 - Join in! Help each other out. Staff will contribute.
 - Also use for private messages with too-specific-to-post questions, code, etc.
 - Staff will also use to post announcements
- Email to `cse401-staff[at]cs` for things that need a followup, not appropriate for ed, ...

Requirements & Grading

- We will have a midterm and final exam
 - It's an important review/reflection part we need
- Roughly:
 - 50% project, done with a partner
 - 25% individual written homework
 - 10% midterm
 - 15% final

We reserve the right to adjust as needed/appropriate

Academic Integrity

- We want a collegial group helping each other succeed!
- But: you must never misrepresent work done by someone else as your own, without proper credit if appropriate, or assist others to do the same
- Read the course policy carefully
- We trust you to behave ethically
 - We have little sympathy for violations of that trust
 - Honest work is the most important feature of a university (or engineering or business or life). Anything less disrespects your instructor, your colleagues, and yourself

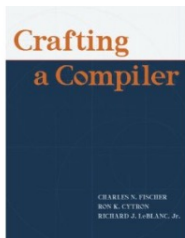
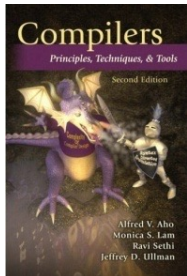
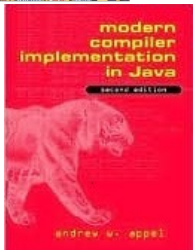
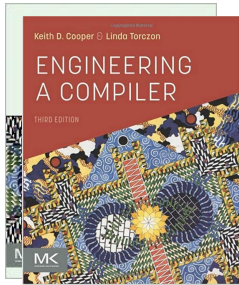
Course Project

- Best way to learn about compilers is to build one!
- Course project
 - MiniJava compiler: classes, objects, etc.
 - Core parts of Java – essentials only
 - Originally from Appel textbook (but you don't need that)
 - Generate executable x86-64 code & run it
 - Completed in steps through the quarter
 - Where you wind up at the end is by far the most important part, but there are intermediate milestones to keep you on schedule and provide feedback at important points
 - Additional work for CSE M 501 students – details later

Project Groups

- You should work in pairs
 - Pick a partner now to work with throughout quarter – we need this info by early next week
 - If you are in CSE M 501 you should pair up with someone else in that group (401 → M 501 switches are possible if it makes sense for individual(s) involved)
 - Partnering over networks works surprisingly well even if you don't intend to hang out together in the labs regularly
- We'll provide accounts on department gitlab server for groups to store and synchronize their work & we'll get files from there for project feedback / grading
 - Anybody new to CSE Gitlab/Git?

Books



- Four good books...
 - Cooper & Torczon, *Engineering a Compiler*. “Official text” & we’ll take some assignments from here. [2nd ed. available free online through UW](#). See syllabus.
 - Appel, *Modern Compiler Implementation in Java*, 2nd ed. MiniJava is from here.
 - Aho, Lam, Sethi, Ullman, “Dragon Book”
 - Fischer, Cytron, LeBlanc, *Crafting a Compiler*

New 3rd edition early this year!
Either 2nd or 3rd ed. should be fine.

And the point is...

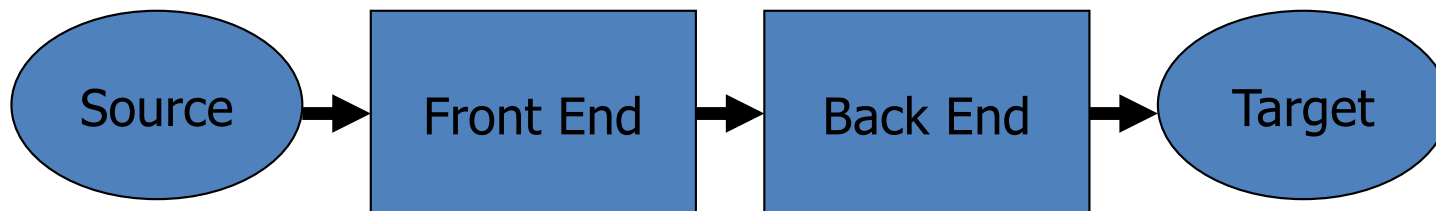
- How do we execute something like this?

```
int nPos = 0;
int k = 0;
while (k < length) {
    if (a[k] > 0) {
        nPos++;
    }
}
```

- Or, more concretely, how do we program a computer to understand and carry out a computation written as text in a file? The computer only knows 1's & 0's: encodings of instructions and data (cf CSE 351)

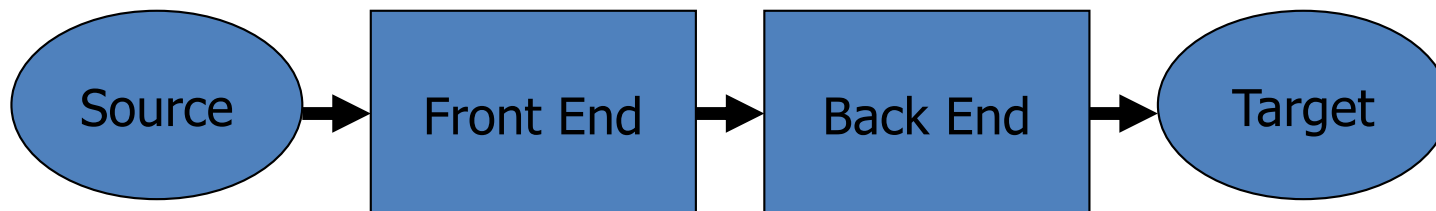
Structure of a Compiler

- At a high level, a compiler has two pieces:
 - Front end: analysis
 - Read source program and discover its structure and meaning
 - Back end: synthesis
 - Generate equivalent target language program



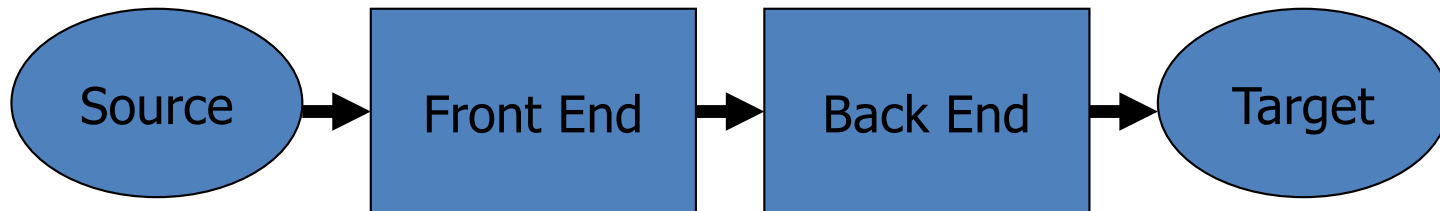
Compiler must...

- Recognize legal programs (& complain about illegal ones)
- Generate correct code
 - Compiler can attempt to improve (“optimize”) code, but must not change behavior (meaning)
- Manage runtime storage of all variables/data
- Agree with OS & linker on target format

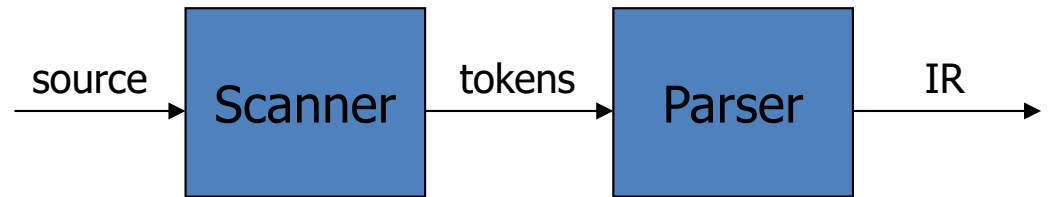


Implications

- Phases communicate using some sort of Intermediate Representation(s) (IR)
 - Front end maps source into IR
 - Back end maps IR to target machine code
 - Often multiple IRs – higher level at first, lower level in later phases



Front End



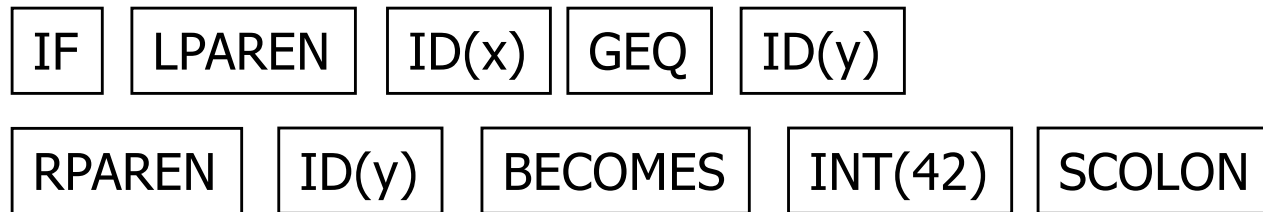
- Usually split into two parts
 - Scanner: Responsible for converting character stream to token stream: keywords, operators, variables, constants, ...
 - Also: strips out white space, comments
 - Parser: Reads token stream; generates IR
 - Either here or shortly after, perform semantics analysis to check for things like type errors, etc.
- Both of these can be generated automatically
 - Use a formal grammar to specify the source language
 - Tools read the grammar and generate scanner & parser (lex/yacc or flex/bison for C/C++, JFlex/CUP for Java)

Scanner Example

- Input text

```
// this statement does very little  
if (x >= y) y = 42;
```

- Token Stream



- Notes: tokens are atomic items, not character strings; comments & whitespace are *not* tokens (in most languages – counterexamples: Python indenting, Ruby and JavaScript newlines)
 - Token objects sometimes carry associated data (e.g., numeric value, variable name)

Parser Output (IR)

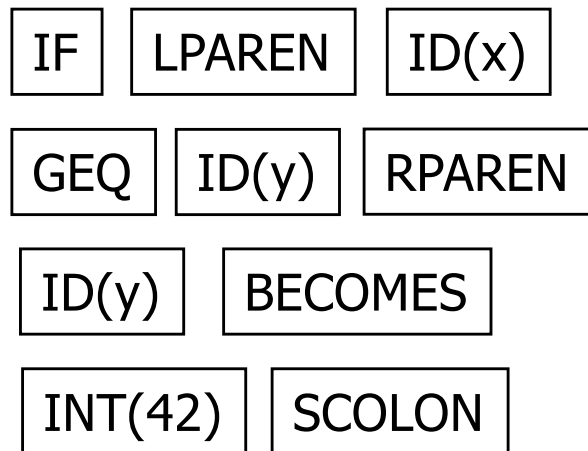
- Given token stream from scanner, the parser must produce output that captures the meaning of the program
- Most common parser output is an abstract syntax tree (AST)
 - Essential meaning of program without syntactic noise
 - Nodes are operations, children are operands
- Many different forms
 - Engineering tradeoffs change over time
 - Tradeoffs (and IRs) can also vary between different phases of a single compiler

Scanner/Parser Example

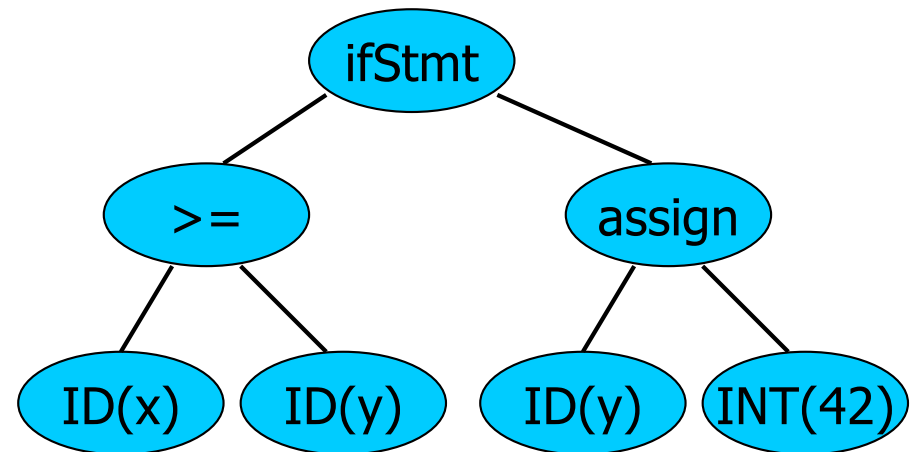
Original source program:

```
// this statement does very little  
if (x >= y) y = 42;
```

- Token Stream



- Abstract Syntax Tree



Static Semantic Analysis

- During or (usually) after parsing, check that the program is legal and collect info for the back end
 - Type checking
 - Verify language requirements like proper declarations, etc.
 - Preliminary resource allocation
 - Collect other information needed by back end analysis and code generation
- Key data structure: Symbol Table(s)
 - Maps names -> meaning/types/details

Back End

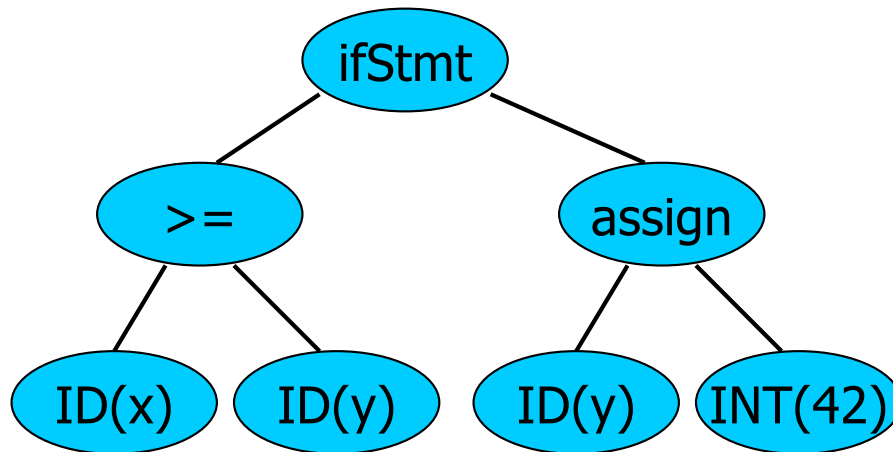
- Responsibilities
 - Translate IR into target code
 - Should produce “good” code
 - “good” = fast, compact, low power (pick some)
 - Should use machine resources effectively
 - Registers
 - Instructions
 - Memory hierarchy

Back End Structure

- Typically two major parts
 - “Optimization” – code improvement – change correct code into semantically equivalent “better” code
 - Examples: common subexpression elimination, constant folding, code motion (move invariant computations outside of loops), function inlining (replace call with body of function)
 - Optimization phases often interleaved with analysis
 - Target Code Generation (machine specific)
 - Instruction selection & scheduling, register allocation
- Usually walk the AST and generate lower-level intermediate code before optimization

The Result

- Input
if (x >= y)
y = 42;



- Output

```
movl 16(%rbp),%edx  
movl -8(%rbp),%eax  
cmpl %eax,%edx  
jl L17  
movl $42,-8(%rbp)  
L17:
```

Why Study Compilers? (1)

- Become a better programmer(!)
 - Insight into interaction between languages, compilers, and hardware
 - Understanding of implementation techniques, how code maps to hardware
 - Better intuition about what your code does
 - Understanding how compilers optimize code helps you write code that is easier to optimize
 - And avoid wasting time doing “optimizations” that the compiler will do better, and avoid “clever” code that confuses the compiler and makes things worse

Why Study Compilers? (2)

- Compiler techniques are everywhere
 - Parsing (“little” languages, program input, scripts,...)
 - Software tools (verifiers, checkers, ...)
 - Database engines, query languages (SQL, ...)
 - Domain-specific languages, ML, data science
 - Text processing
 - Tex/LaTeX -> dvi -> Postscript -> pdf
 - Hardware: VHDL; model-checking tools
 - Mathematics (Mathematica, Matlab, SAGE)

Why Study Compilers? (3)

- Fascinating blend of theory and engineering
 - Lots of beautiful theory around compilers
 - Parsing, scanning, static analysis
 - Interesting engineering challenges and tradeoffs, particularly in optimization (code improvement)
 - Ordering of optimization phases
 - What works for some programs can be bad for others
 - Plus some very difficult problems (NP-hard or worse)
 - E.g., register allocation is equivalent to graph coloring
 - Need to come up with “good enough” approximations / heuristics

Why Study Compilers? (4)

- Draws ideas from many parts of CSE
 - AI: Greedy algorithms, heuristic search
 - Algorithms: graphs, dynamic programming, approximation
 - Theory: Grammars, DFAs and PDAs, pattern matching, fixed-point algorithms
 - Systems: Allocation & naming, synchronization, locality
 - Architecture: pipelines, instruction set use, memory hierarchy management, locality

Why Study Compilers? (5)

- You might even write a compiler some day!
- You *will* write parsers and interpreters for little languages, if not bigger things
 - Command languages, configuration files, XML, JSON, network protocols, ...
- And if you like working with compilers and are good at it there are many jobs available...
 - Novel languages / architectures for ML, massive data science, etc. need effective implementations

Any questions?

- Your job is to ask questions to be sure you understand what's happening and to slow me down
 - Otherwise, I'll barrel on ahead 😊

Coming Attractions

- Quick review of formal grammars
- Lexical analysis – scanning & regular expressions
 - Background for first part of the project
- Followed by parsing ...
- Start reading: ch. 1, 2.1-2.4
 - Entire 2nd ed. book available through Safari Online to UW community – see syllabus for link

Before next time...

- Familiarize yourself with the course web site
- Read syllabus and academic integrity policy
- Find a partner!
 - And meet other people in the class too!! 😊