CSE P 501 – Compilers

Overview and Administrivia

Hal Perkins

Autumn 2025

Agenda

- Introductions
- What's a compiler?
- Administrivia

Agenda

- Introductions
- What's a compiler?
- Administrivia

Who: Course staff

- Instructor: Hal Perkins: UW faculty for quite a while now; veteran of many compiler courses (among other things)
- Teaching Assistant: Alexander Metzger , CSE grad student
  - Plus help from the staff of the related CSE401/M501 compiler classes
- Office hours: Alexander, Wed. before classes, 5:30-6:20, CSE2 153; Hal, after class – lecture room
  - Can start Wed. at 5:00 – worth doing?
  - Last time we did CSE P 501 we had weekend zoom office hours Sat. afternoon. Can do that again or some other time. (Most assignments due the day before class.) What would help?
- Get to know us – we're here to help you succeed!

Who: You!

- About 50 people in the class
  - With luck we'll all get a chance to get to know each other over the quarter
- Assumption (based on past experience) is that this is a first complete compiler course for most everyone – sound about right?
  - Will adjust as needed based on background, but we want to have comprehensive coverage of compilers

Credits

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

Agenda

- Introductions
- What’s a compiler?
- Administrivia

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 precisely, 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

Interpreters & Compilers

- Programs can be compiled or interpreted (or sometimes both)
- Compiler
  - A program that translates a program from one language (the source) to another (the target)
    - Languages are sometimes even the same(!)
- Interpreter
  - A program that reads a source program and produces the results of executing that program on some input as it reads it

Common Issues

- Compilers and interpreters both must read the input – a stream of characters – and "understand" it: front-end *analysis* phase

w h i l e (k < l e n g t h) { <nl> <tab> i f (a [k] > 0
) <nl> <tab> <tab>{ n P o s + + ; } <nl> <tab> }

Compiler

- Read and analyze entire program
- Translate to semantically equivalent program in another language
  - Presumably easier or more efficient to execute
- Offline process
- Tradeoff: compile-time overhead (preprocessing) vs execution performance

Typically implemented with Compilers

- FORTRAN, C, C++, C#, many other programming languages, (La)TeX, VHDL, many others
- Particularly appropriate if significant optimization wanted/needed

Interpreter

- Interpreter
  - Typically implemented as an “execution engine”
  - Program analysis interleaved with execution:

```
running = true;
while (running) {
   analyze next statement;
   execute that statement;
}
```

  - Usually requires repeated analysis of individual statements (particularly in loops, functions)
    - Hybrid approaches can avoid some of this overhead
  - But: immediate execution, good debugging, interactive, …

Often implemented with interpreters

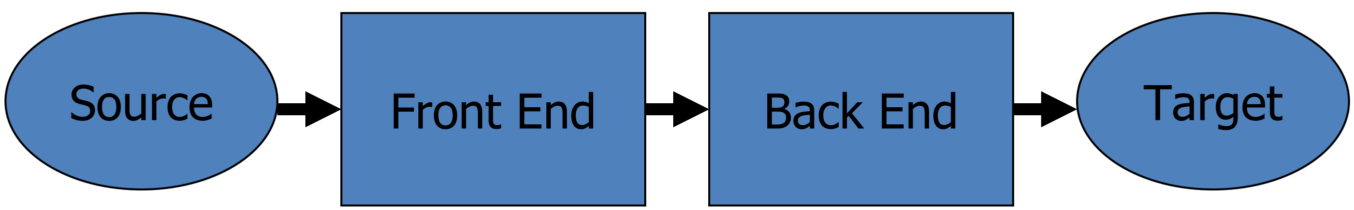
- JavaScript, PERL, Python, Ruby, awk, sed, shells (bash), Racket/Scheme/Lisp/ML/OCaml, SQL (databases), postscript/pdf, machine simulators
- Particularly efficient if interpreter overhead is low relative to execution cost of individual statements
  - But even if not (machine simulators), flexibility, immediacy, or portability may be worth it

Hybrid approaches

- Compiler generates intermediate language, e.g. compile Java source to Java Virtual Machine .class files (byte codes), then:
  - Interpret byte codes directly, or
  - Compile some or all byte codes to native code
    - Variation: Just-In-Time compiler (JIT) – detect hot spots & compile on the fly to native code
- Widely use for Java, C#, JavaScript, many functional and other languages (Haskell, ML, Racket, Ruby), …

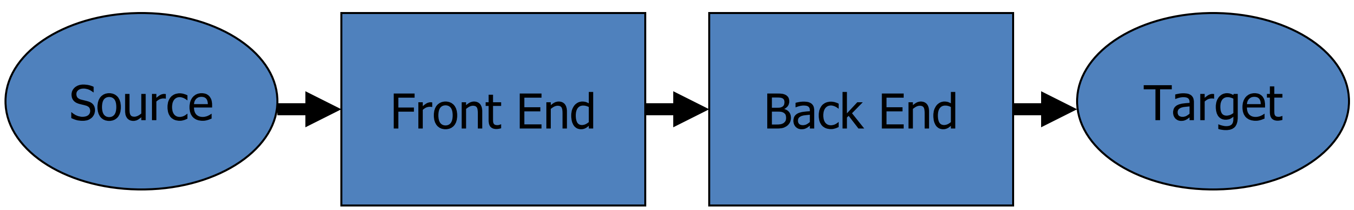
Structure of a Compiler

- 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



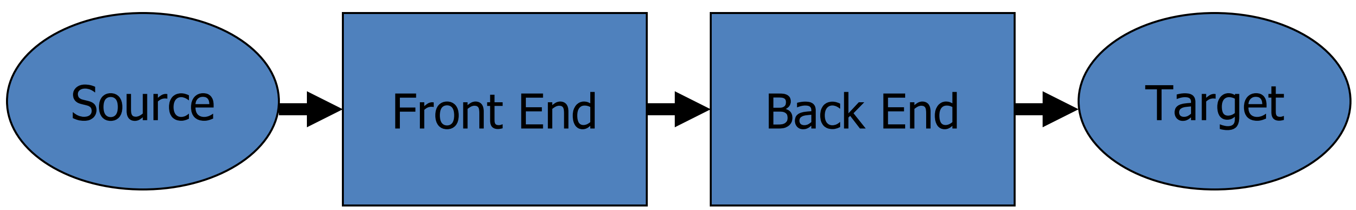
Compiler must…

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

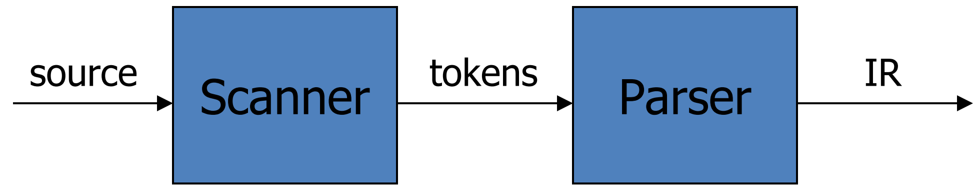


Implications

- 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: Convert character stream to token stream: keywords, operators, variables, constants, …
    - Also: strip out white space, comments
  - Parser: Read token stream; generates IR (AST or other)
    - Either here or right after, check for semantics rules like type errors that are not captured in the parser grammar
- Scanner & parser 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, equivalent tools for almost all major languages)

Scanner Example

- Input text

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

- Token Stream

IF	LPAREN	ID(x)	GEQ	ID(y)
RPAREN	ID(y)	BECOMES	INT(42)	SCOLON

– Notes: tokens are atomic items, ***not*** character strings; comments & whitespace are *not* tokens (in most languages – counterexamples: Python indenting, Ruby and JavaScript newlines)
  - Tokens can carry associated data (e.g., int value, variable name, location in source program, …)

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 often vary between different phases of a single compiler

Scannner/Parser Example

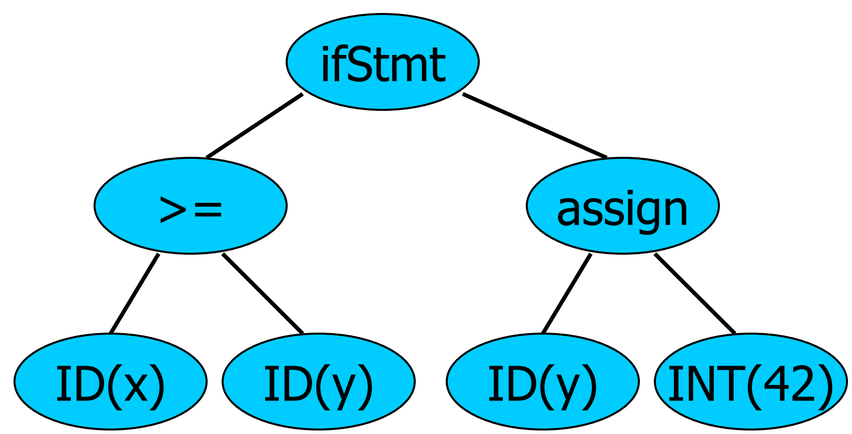
Original source program:

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

- Token Stream

IF LPAREN ID(x)
GEQ ID(y) RPAREN
ID(y) BECOMES
INT(42) SCOLON

- Abstract Syntax Tree



Static Semantic Analysis

- During or (usually) after parsing, check that the program is legal and collect info for the back end
- Context-dependent checks that cannot be captured in a context-free grammar
  - Type checking (e.g., int x = 42 + true, number and types of arguments in method call, …)
  - Check language requirements like proper declarations, etc.
  - Preliminary resource allocation
  - Collect other information needed for back end analysis and code generation
- Key data structure: Symbol Table(s)
  - Maps names -> meanings/types/details

Back End

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

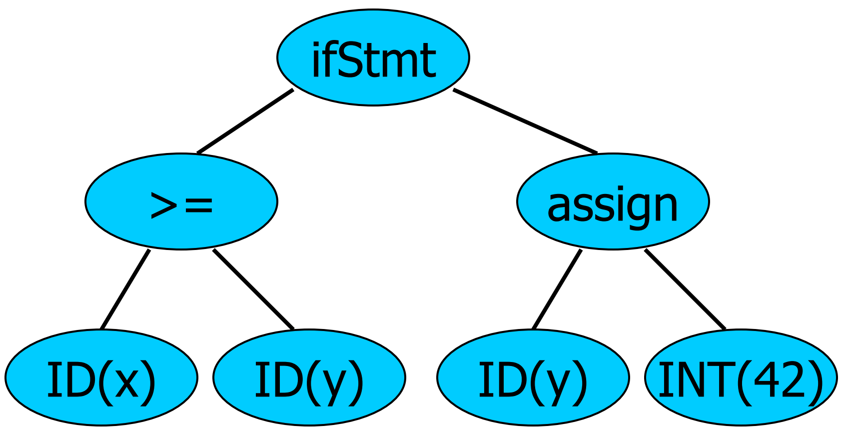
Back End Structure

- 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 function body)
    - 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
    - Avoid wasting time on source "optimizations" that the compiler ~~can~~ will do better; avoid "clever" code that confuses the compiler and makes things worse

Why Study Compilers? (2)

- Compiler techniques are everywhere
  - Parsing ("little" languages, interpreters, XML)
  - 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: graph, 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, semi-structured data, …

- And if you like working with compilers and are good at it there are many jobs available…
  - Novel languages / architectures for ML/AI, massive data science, etc. need effective implementations

Some History (1)

- 1950's. Existence proof
  - FORTRAN I (1954) – competitive with hand-optimized code
- 1960's
  - New languages: ALGOL, LISP, COBOL, SIMULA
  - Formal notations for syntax, esp. BNF
  - Fundamental implementation techniques
    - Stack frames, recursive procedures, etc.

Some History (2)

- 1970's
  - Syntax: formal methods for producing compiler front-ends; many, many theorems
- Late 1970's, 1980's
  - New languages (functional; object-oriented, especially Smalltalk – a direct ancestor of Java)
  - New architectures (RISC machines, parallel machines, memory hierarchy issues)
  - More attention to back-end issues

Some History (3)

- 1990s
  - Techniques for compiling objects and classes, efficiency in the presence of dynamic dispatch and small methods (Self – precursor of JavaScript, Smalltalk; techniques now common in JVMs, etc.)
  - Just-in-time compilers (JITs)
  - Compiler technology critical to effective use of new hardware (RISC, parallel machines, complex memory hierarchies)

Some History (4)

- 21st Century:
  - Compilation techniques in many new places
    - Software static analysis, verification, security
  - Phased compilation – blurring the lines between “compile time” and “runtime”
  - Dynamic languages – e.g., JavaScript, …
  - Domain-specific languages (DSL)
  - Custom hardware for applications
  - Etc. …

Some History (5)

- 21st Century (more):
  - Memory models, concurrency, multicore, …
  - Full stack proofs/verification; secure OS/compilers
  - Language implementations for novel, heterogenous hardware architectures (dealing with the end of Moore's law, etc.), app-specific processors (GPUs), AI/ML
    - How do we program these things? Need software tools – languages, compilers, …
  - Etc. etc.

Compiler (and related) Turing Awards

- 1966 Alan Perlis
- 1971 John McCarthy
- 1972 Edsger Dijkstra
- 1974 Donald Knuth
- 1976 Michael Rabin and Dana Scott
- 1977 John Backus
- 1978 Bob Floyd
- 1979 Ken Iverson
- 1980 Tony Hoare
- 1984 Niklaus Wirth
- 1987 John Cocke
- 1991 Robin Milner
- 2001 Ole-Johan Dahl and Kristen Nygaard
- 2003 Alan Kay
- 2005 Peter Naur
- 2006 Fran Allen
- 2008 Barbara Liskov
- 2013 Leslie Lamport
- 2018 John Hennessy & David Patterson
- 2020 Al Aho & Jeffrey Ullman

Agenda

- Introductions
- What's a compiler?
- Administrivia

What's in CSE P 501?

- In past years most P 501 students either have never taken a compiler course or what was covered was a mixed bag, so…
- We will cover the basics, but fairly quickly…
- Then coverage of more advanced topics
- If you have some background, some of this may be review, but everyone will probably pick up new things (ask instructor if not sure)

Expected background

- Assume undergraduate courses or equiv. in:
  - Data structures and algorithms
    - Linked lists, trees, hash tables, dictionaries, graphs
  - Machine organization
    - Assembly-level programming/instruction set of some architecture (not necessarily x86-64)
  - Formal languages & automata
    - Regular expressions, NFAs/DFAs, context-free grammars, maybe a little parsing
- We will review basics and gaps can be filled in but might take some extra time/work if needed

CSE P 501 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 won't need that)
  - Generate executable x86-64 code & run it
  - Every legal MiniJava program is also legal regular Java – compare results from your project with javac/java

Project Scope

- Goal: large enough to be interesting and capture key concepts; small enough to do in 10 weeks
- Completed in steps through the quarter
  - Where you wind up at the end is the most important
  - Intermediate milestone deadlines to keep you on schedule and provide feedback at important points
  - Evaluation is weighted towards final result (~50% of the total) but milestone results count
- Core requirements, then open-ended if you have time for extensions and are interested

Project Implementation

- Default is Java 21 with JFlex, CUP scanner/parser tools
  - Choice of editors/environments up to you
- Somewhat open to alternatives – check with course staff – but you assume risk of the unknown
  - Have had successful past projects using C#, F#, Haskell, ML, others (even Python & Ruby!)
  - You need to be sure there are Lex/Yacc, Flex/Bison work-alike compiler tools available
  - Your compiler has to "work" the same as the regular ones (startup, command options, etc.) and you need to provide specific instructions so staff can run tests and evaluate
  - Course staff will help as best we can but no guarantees

Project Groups & Repositories

- You should work in groups of two
  - Pick a partner now to work with throughout quarter
    - How? Mingle during breaks, after class, discussion board, …
  - Have had some people do the project solo, but easy to underestimate effort needed. Very helpful to have partner to talk to about details. Pairs very very strongly recommended.
- All groups *must* use course repositories on CSE GitLab server to store their projects. We'll access files from there for evaluation (& to help with project during office hours – especially useful if set up remote zoom OH)
- By early next week, fill out partner info form linked on course web so we can set up groups and repositories
  - Would like this by noon next Thur. and will set up repos then

Requirements & Grading

- Roughly
  - 55% project
  - 25% individual written homework
  - 20% exam (late in quarter, preferable separate time not during regular class)
    - Let’s look at the calendar now to figure out when

  We reserve the right to adjust as needed

- Deadlines – would like to be able to hand out sample solutions next class after assignments are due, but not sure how best to time this. Thoughts?

Academic Integrity

- We want a collegial group helping each other succeed!
- But: you must never misrepresent work done by someone (or something) else as your own, without proper credit when appropriate, or assist others to do the same
  - Do not attempt to bypass learning by avoiding work or help others do the same
- Read the course policy on the website carefully
- We trust you to behave ethically
  - We have little sympathy for violations of that trust
- Honest work is the foundation of your university work (and engineering and business and life). Anything less disrespects your teachers, your colleagues, and yourself
- If in doubt about whether something is ok, ask.

Lectures

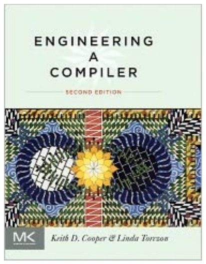
- Wednesdays, 6:30-9:20
  - Recorded, but not intended as substitute for attending
- Lecture slides posted on course calendar by mid-afternoon before each class (usually night before)

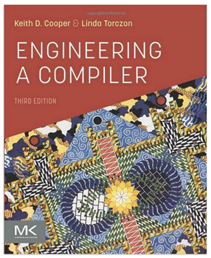
- Strongly recommend no laptops / devices in class unless you are actually using a tablet to take notes – devices are distracting and addictive
  - Something confusing? Don’t search; ask a question!
    - Your colleagues will be grateful! 🙂

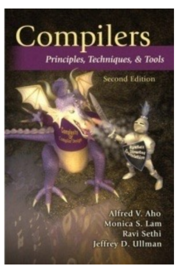
Staying in touch

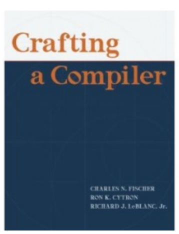
- Course web site (www.cs.uw.edu/csep501)
- Discussion board – ed
  - For (almost) anything related to the course
  - Join in! Help each other out. Staff will contribute.
  - Also use for private messages with too-specific-to-post questions, code, etc.
  - Staff will also use to post announcements
- Gradescope written assignment submission and regrade requests / feedback questions
- Email to csep501-staff[at]cs for project feedback questions, unexpected or personal situations, things that need a followup not appropriate for ed, …

Books









Four good books – use at least one…

- Cooper & Torczon, *Engineering a Compiler*, 2^{nd} or 3^{rd} edition "Official text" & we'll take some assignment questions from here. 2^{nd} ed available free online through UW Library Safari books login. See syllabus.
- Appel, *Modern Compiler Implementation in Java*, 2^{nd} ed. MiniJava is from here.
- Aho, Lam, Sethi, Ullman, "Dragon Book"
- Fischer, Cytron, LeBlanc, *Crafting a Compiler*

Any questions?

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

Coming Attractions

- Quick review of formal grammars
- Lexical analysis – scanning
  - Background for first part of the project
- Followed by parsing …

- Start reading: ch. 1, 2.1-2.4 in EAC or corresponding chapters in other books

Todo by next week

- Familiarize yourself with the course web site
- Read syllabus and academic integrity policy

- HW1 – regexps/DFAs – due next Tuesday night
  - Gradescope accounts will be set up in the next couple of days for hw submission

- Find a partner!
  - And meet other people in the class too!! ☺️
  - Partner form due next Thursday noon