CSE 401 – Compilers

Overview and Administrivia

Hal Perkins

Winter 2015

Agenda

- Introductions
- Administrivia
- What's a compiler?
- Why you want to take this course
- Some other time: a bit of history

Who: Course staff

- Instructor:
  - Hal Perkins: UW faculty for quite a while now; have taught various compiler courses many times
- TAs:
  - Meghan Cowan, Nat Mote, David Wong
- Office hours will be figured out ASAP
- Get to know us – we're here to help you succeed!

Credits

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

So whadda ya know?

- Official prerequisites:
  - CSE 332 (data abstractions)
    - and therefore CSE 311 (Foundations)
  - CSE 351 (hardware/software interface)
- We think this is the first year that everyone has seen the x86-64 version of CSE 351. Is that right?
- Also 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
  - Arrive punctually and pay attention (& take notes!)
  - If doing so doesn't save you time, one of us is messing up!
- Sections: additional examples and exercises plus project and tools background
- Additional project and other material posted

Staying in touch

- Course web site
- Discussion board
  - For anything related to the course
  - Join in! Help each other out. Staff will contribute.
- Mailing list
  - You are automatically subscribed if you are registered
  - Will keep this fairly low-volume; limited to things that everyone needs to read

Requirements & Grading

- Roughly
  - 50% project, done with a partner
  - 15% individual written homework
  - 15% midterm exam (probably Wed. Feb. 11)
  - 20% final exam

  We reserve the right to adjust as needed

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
  - I have little sympathy for violations of that trust
  - Honest work is the most important feature of a university (or engineering or business). Anything less disrespects your instructor, your colleagues, and yourself

CSE 401 Course Project

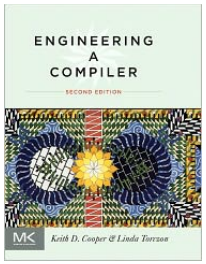
- Best way to learn about compilers is to build one
- Course project
  - MiniJava compiler: classes, objects, etc.
    - Core parts of Java – essentials only
    - Originally from Appel textbook (but you won't need that)
  - Generate executable x86-64 code & run it
  - Completed in steps through the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

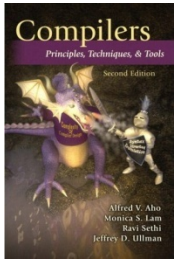
Project Groups

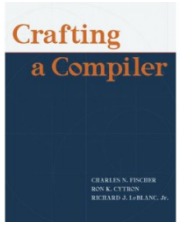
- You should work in pairs
  - Pick a partner now to work with throughout quarter

- We'll provide accounts on a new department git server (gitlab) for groups to store and synchronize their work
  - And we'll go over how to use it in sections

Books







- Four good books; will put on reserve in the engineering library:
  - Cooper & Torczon, *Engineering a Compiler.* "Official text" 1st edition should be ok too.
  - Appel, *Modern Compiler Implementation in Java*, 2nd ed. MiniJava is from here.
  - Aho, Lam, Sethi, Ullman, "Dragon Book"
  - Fischer, Cytron, LeBlanc, *Crafting a Compiler*

And the point is…

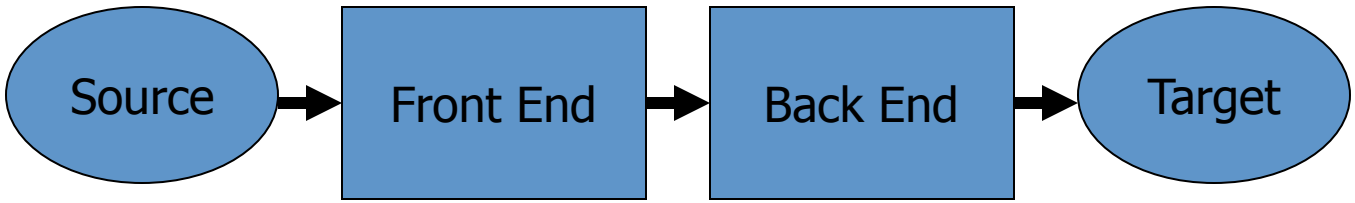
- How do we execute something like this?

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

- The computer only knows 1's & 0's - i.e., encodings of instructions and data

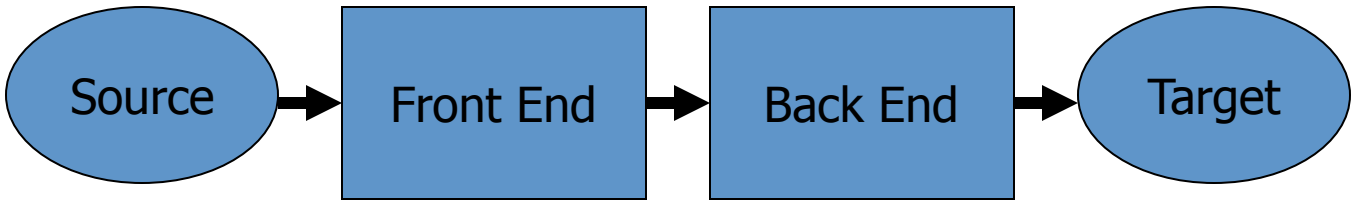
Structure of a Compiler

- 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



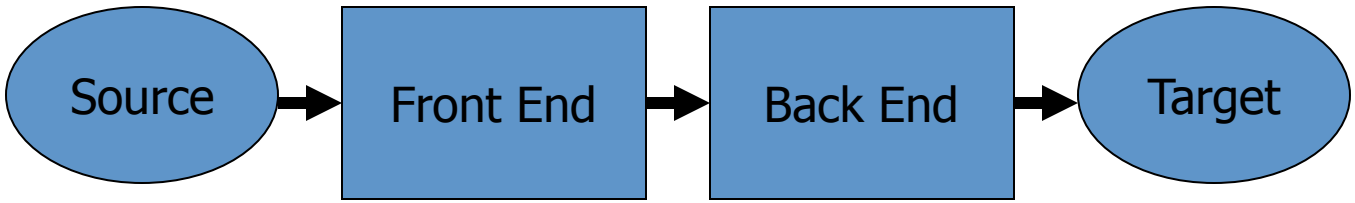
Compiler must…

- 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

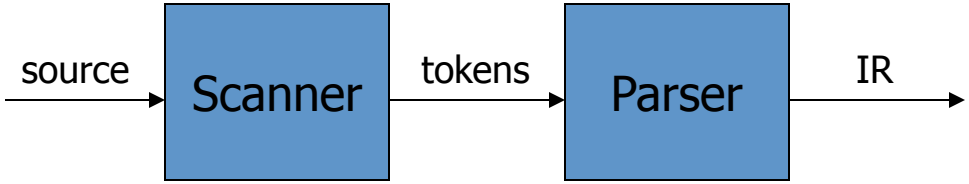


Implications

- 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



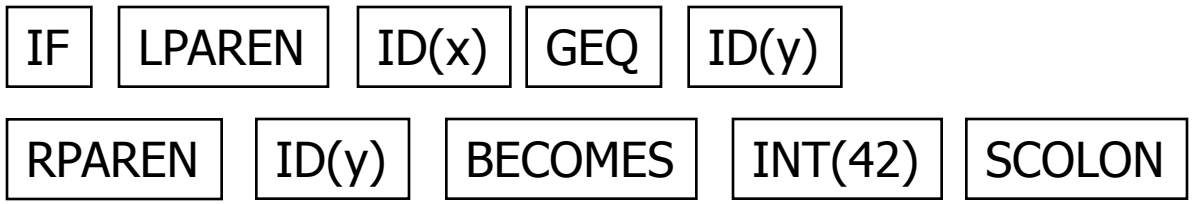
- Usually split into two parts
  - Scanner: Responsible for converting character stream to token stream: keywords, operators, variables, constants, …
    - Also: strips out white space, comments
  - Parser: Reads token stream; generates IR
    - Either here or shortly after, perform semantics analysis 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 newlines)
    - Tokens may carry associated data (e.g., int value, variable name)

Parser Output (IR)

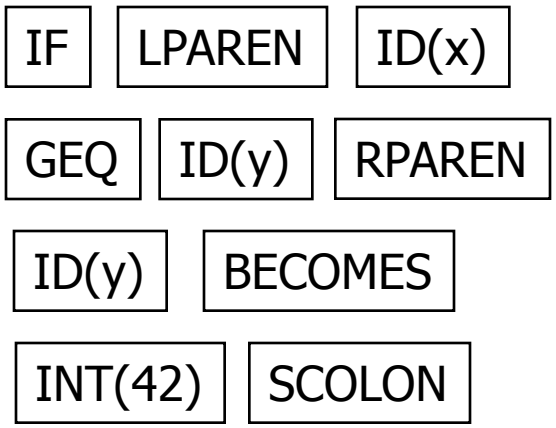
- Given token stream from scanner, the parser must produce output that captures the meaning of the program
- Most common output from a parser is an abstract syntax tree
  - Essential meaning of program without syntactic noise
  - Nodes are operations, children are operands
- Many different forms
  -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

Parser Example

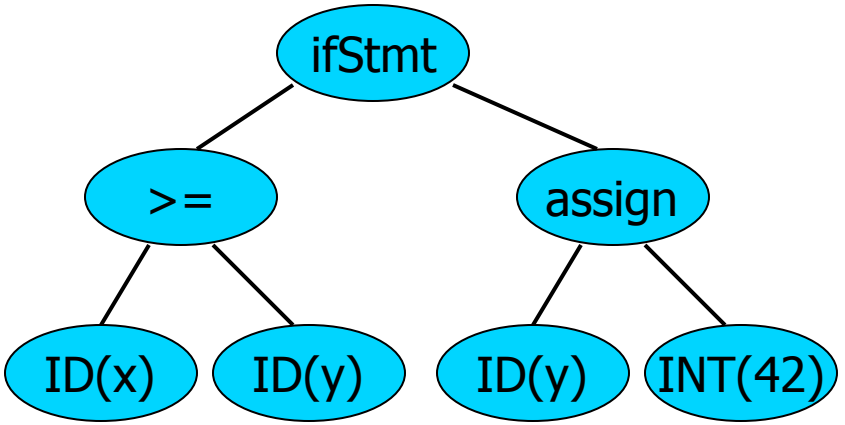
Original source program:

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

- Token Stream



- Abstract Syntax Tree



Static Semantic Analysis

- During or after parsing, check that the program is legal and collect info for the back end
  - Type checking
  - Check 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 machine code
  - Should produce "good" code
    - "good" = fast, compact, low power (pick some)
    - Optimization phase translates correct code into semantically equivalent "better" code
  - Should use machine resources effectively
    - Registers
    - Instructions
    - Memory hierarchy

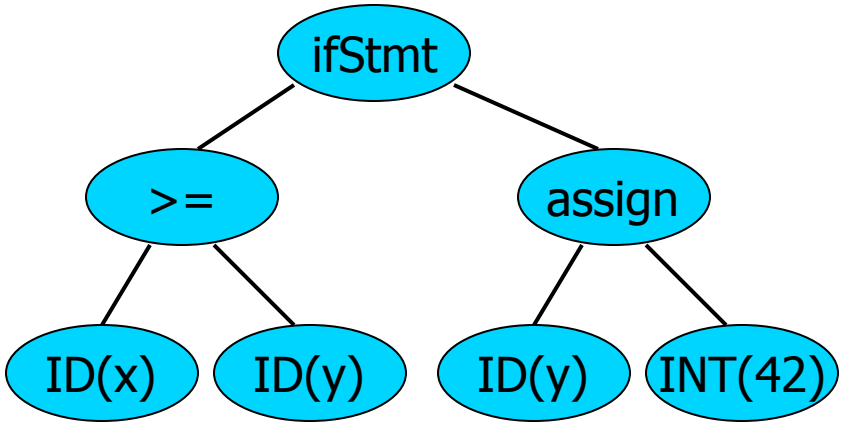
Back End Structure

- Typically split into two major parts
  - "Optimization" – code improvements
    - Examples: common subexpression elimination, constant folding, code motion (move invariant computations outside of loops)
    - Optimization phases often interleaved with analysis
  - Target Code Generation (machine specific)
    - Instruction selection & scheduling, register allocation
  - Usually walk the AST to generate lower-level intermediate code before optimization

The Result

- Input

```
if (x >= y)
    y = 42;
```



- Output

```
    mov   eax,[ebp+16]
    cmp   eax,[ebp-8]
    jl      L17
    mov    [ebp-8],42
L17:
```

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

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++, COBOL, many other programming languages, (La)TeX, SQL (databases), 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)
    - But hybrid approaches can avoid some of this overhead
  - But: immediate execution, good debugging/interaction, etc.

Often implemented with interpreters

- Javascript, PERL, Python, Ruby, awk, sed, shells (bash), Scheme/Lisp/ML/OCaml, 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 byte code intermediate language, e.g. compile Java source to Java Virtual Machine .class files, 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
- Also wide use for Javascript, many functional and other languages (Haskell, ML, Ruby), C# and Microsoft Common Language Runtime, others

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 as well or better – particularly if you don't try to get too clever

Why Study Compilers? (2)

- Compiler techniques are everywhere
  - Parsing ("little" languages, interpreters, XML)
  - Software tools (verifiers, checkers, …)
  - Database engines, query languages
  - AI, etc.: domain-specific languages
  - 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 algorithms, dynamic programming, approximation algorithms
  - 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, network protocols, …

- And if you like working with compilers and are good at it there are many jobs available…

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 theorems
- Late 1970's, 1980's
  - New languages (functional; object-oriented - Smalltalk)
  - 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, Smalltalk – 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)

- Last decade
  - Compilation techniques in many new places
    - Software analysis, verification, security
  - Phased compilation – blurring the lines between "compile time" and "runtime"
    - Using machine learning techniques to control optimizations(!)
  - Dynamic languages – e.g., JavaScript, …
  - The new 800 lb gorilla - multicore

Compiler (and related) Turing Awards

- 1966 Alan Perlis
- 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

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 ☺

Before next time…

- If you are trying to add the class please sign the sheet before leaving today
- Familiarize yourself with the course web site
- Read syllabus and academic integrity policy
- Fill in the office hours doodle linked from the web site
- Post a followup message on the discussion board

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