CSE 401 – Compilers

Overview and Administrivia

Hal Perkins

Autumn 2011

Credits

- Some direct ancestors of this quarter:
  - UW CSE 401 (Chambers, Snyder, Notkin…)
  - 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, …)

Agenda

- Introductions
- Administrivia
- What's a compiler?
- Why you want to take this course
- & a little history if time

CSE 401 Personnel

- Instructor: Hal Perkins
  - CSE 548; perkins[at]cs
  - Office hours: tbd + dropins, etc.
- TAs: Sam Fout, Evan Herbst
  - Office hours, etc. tbd.
- You!!!

So whadda ya know?

- The revenge of the new core curriculum
- Official prereq: (326 & 378) | (332 & 351)
  - E.g., data structures and machine organization
- Who took what?
  - CSE 321/322 vs CSE 311/312
  - CSE 326 vs CSE 332
  - CSE 378 vs CSE 351
  - CSE 341 (now optional)

Course Meetings

- Lectures
  - MWF 12:30-1:20 here (More 230)
- Sections Thursdays
  - AA: 8:30, AB: 9:30, both in More 221
  - No sections this week – not far enough along yet

Communications

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

Requirements & Grading

- Roughly
  - 40% project
  - 15% individual written homework
  - 15% midterm exam (date tbd*)
  - 25% final exam
  - 5% other

  We reserve the right to adjust as needed
- *Midterm date: Nov. 4 (day after Thur. section)? Or Nov. 7 (following Monday)? Preferences?

CSE 401 Course Project

- Best way to learn about compilers is to build (at least parts of) one
- Course project
  - Mini Java compiler: classes, objects, etc.
    - But cut down to essentials
  - Generate executable x86(-64) code & run it
  - Completed in steps through the quarter
    - Intermediate steps to keep you on schedule but where you wind up at the end is major part

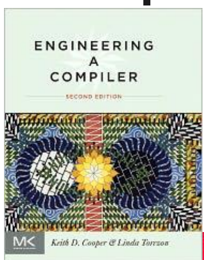
Project Groups

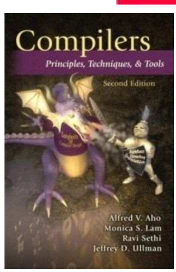
- You should work in pairs
  - Pair programming strongly encouraged
- Space for group SVN repositories & other shared files provided
- Pick partners soon (end of this week or by beginning of next)

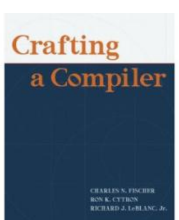
Academic Integrity

- We want a cooperative group working together to do great stuff!
- But: you must never misrepresent work done by someone else as your own, without proper credit
- Know the rules – ask if in doubt or if tempted

Books







- Four good books, all on Eng. Lib. Reserve:
  - Cooper & Torczon, *Engineering a Compiler.* "Official text" New edition this year, but first is still good.
  - Appel, *Modern Compiler Implementation in Java*, 2nd ed. MiniJava adapted from here.
  - Aho, Lam, Sethi, Ullman, "Dragon Book", 2nd ed (but 1st ed is also fine)
  - Fischer, Cytron, LeBlanc, *Crafting a Compiler*

And the point is…

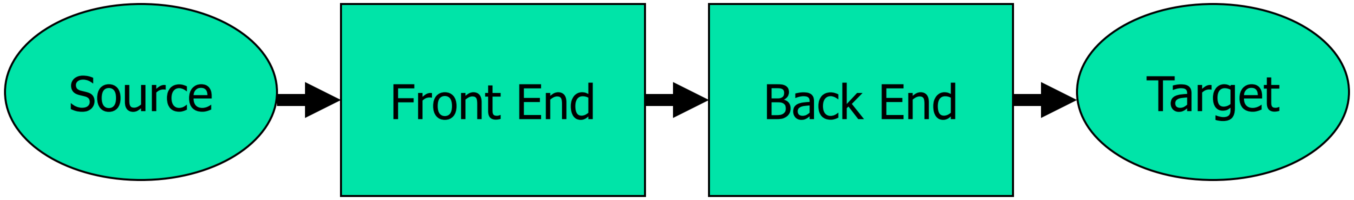
- How do we execute this?

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

- The computer only knows 1's & 0's

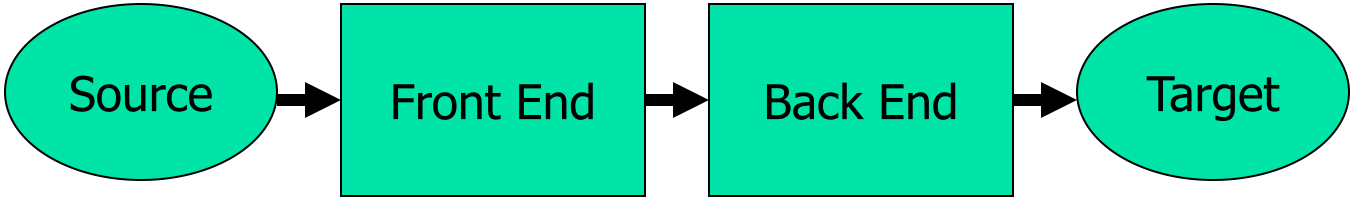
Structure of a Compiler

- First approximation
  - Front end: analysis
    - Read source program and understand its structure and meaning
  - Back end: synthesis
    - Generate equivalent target language program



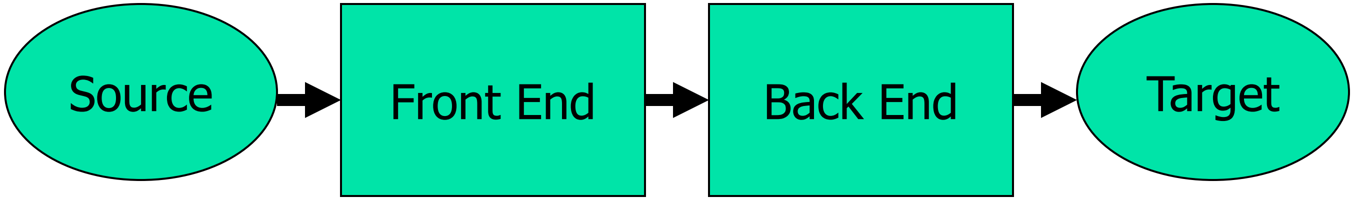
Compiler must…

- recognize legal programs (& complain about illegal ones)
- generate correct code
- manage runtime storage of all variables/data
- agree with OS & linker on target format

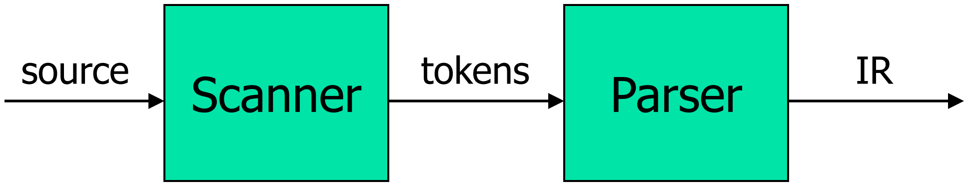


Implications

- Need 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
    - Also: strips out white space, comments
  - Parser: Reads token stream; generates IR
- Both of these can be generated automatically
  - Source language specified by a formal grammar
  - Tools read the grammar and generate scanner & parser (either table-driven or hard-coded)

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 – counterexample: Python)

Parser Output (IR)

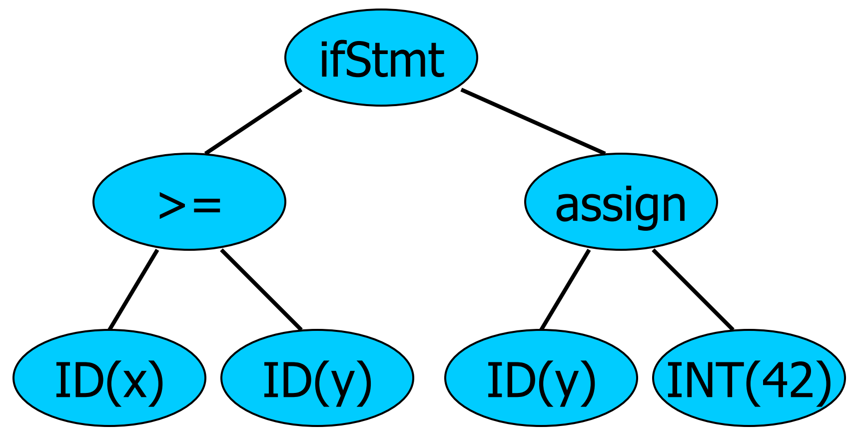
- Many different forms
  - Engineering tradeoffs have changed over time (e.g., memory is (almost) free these days)
- Common output from a parser is an abstract syntax tree
  - Essential meaning of the program without the syntactic noise

Parser Example

- Token Stream Input

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

- Abstract Syntax Tree



Static Semantic Analysis

- During or (more common) after parsing
  - 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)
  - Should use machine resources effectively
    - Registers
    - Instructions
    - Memory hierarchy

Back End Structure

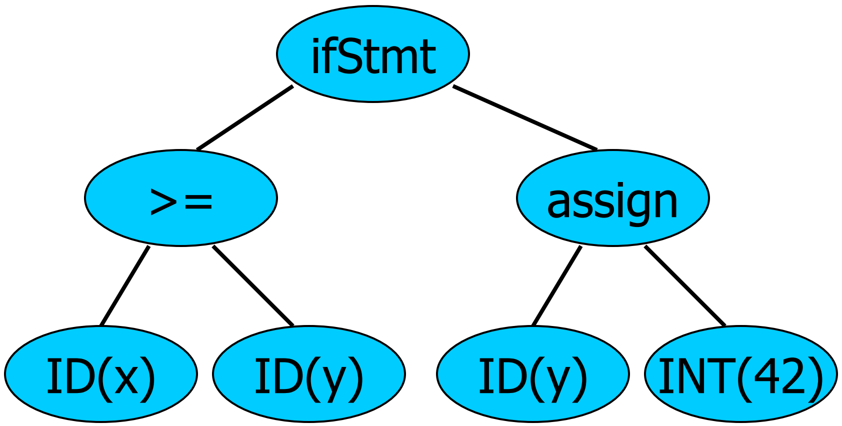
- Typically split into two major parts
  - "Optimization" – code improvements
  - Target Code Generation (machine specific)
    - Instruction selection & scheduling
    - Register allocation
  - Usually walk the AST to generate lower-level intermediate code before optimization

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

- Compiler
  - A program that translates a program from one language (the *source*) to another (the *target*)
- 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, other programming languages, (La)TeX, SQL (databases), VHDL, many others
- Particularly appropriate if significant optimization wanted/needed

Interpreter

- Interpreter
  - 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: immediate execution, good debugging & interaction, etc.

Often implemented with interpreters

- Javascript, PERL, Python, Ruby, awk, sed, shells (bash), Scheme/Lisp/ML, 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 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
  - What is all that stuff in the debugger anyway?
  - Better intuition about what your code does

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)

Why Study Compilers? (3)

- Fascinating blend of theory and engineering
  - Direct applications of theory to practice
    - Parsing, scanning, static analysis
  - Plus some very difficult problems (NP-hard or worse)
    - Resource allocation, "optimization", etc.
    - 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, …

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 ☺

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