CSE P 501 – Compilers

Overview and Administrivia

Hal Perkins

Autumn 2011
Credits

- Some direct ancestors of this course
  - UW CSE 401 (Chambers, Snyder, Notkin...)
  - Cornell CS 412-3 (Teitelbaum, Perkins)
  - Rice CS 412 (Cooper, Kennedy, Torczon)
  - Many other compiler courses, some papers
  - Many books (Appel; Cooper/Torczon; Aho, [[Lam,] Sethi,] Ullman [Dragon Books]; Fischer, Cryton, LeBlanc; Muchnick, ...)

10/4/2011

© 2002-11 Hal Perkins & UW CSE
Agenda

- Introductions
- What’s a compiler?
- Administrivia
CSE P 501 Personnel

- Instructor: Hal Perkins
  - CSE 548; perkins@cs
  - Office hours: after class + drop in when you’re around + appointments
    - (& before class if I’m not swamped)

- TA: Soumya Vasisht
  - vasisht@cs
  - Office hours: Tue 5:30-6:20; location tba
And the point is...

- Execute this!

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

- How? Computers only know 1’s and 0’s
Interpreters & Compilers

- Interpreter
  - A program that reads an source program and produces the results of executing that program

- Compiler
  - A program that translates a program from one language (the source) to another (the target)
Common Issues

- Compilers and interpreters both must read the input – a stream of characters – and “understand” it; *analysis*

```c
while (k < length) {
  if (a[k] > 0)
    for (nPos++)
  }
```
Interpreter

- Interpreter
  - Execution engine
  - Program execution interleaved with analysis

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

- Usually need repeated analysis of statements (particularly in loops, functions)
- But: immediate execution, good debugging & interaction
Compiler

- Read and analyze entire program
- Translate to semantically equivalent program in another language
  - Presumably easier to execute or more efficient
  - Should “improve” the program in some fashion
- Offline process
  - Tradeoff: compile time overhead (preprocessing step) vs execution performance
Typical Implementations

- **Compilers**
  - FORTRAN, C, C++, Java, COBOL, etc. etc.
  - Strong need for optimization in many cases

- **Interpreters**
  - PERL, Python, Ruby, awk, sed, shells, Scheme/Lisp/ML, postscript/pdf, Java VM
  - Particularly effective if interpreter overhead is low relative to execution cost of individual statements
Hybrid approaches

- Classic example: Java
  - Compile Java source to byte codes – Java Virtual Machine language (.class files)
  - Execution
    - Interpret byte codes directly, or
    - Compile some or all byte codes to native code
      - Just-In-Time compiler (JIT) – detect hot spots & compile on the fly to native code – standard these days

- Variations used for .NET (compile always) & implementations of dynamic and functional languages, e.g., JavaScript, Haskell
Why Study Compilers? (1)

- Become a better programmer(!)
  - Insight into interaction between languages, compilers, and hardware
  - Understanding of implementation techniques
  - 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
  - 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)

- 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
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, ...
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 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

- Normally 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**

- **Notes:** tokens are atomic items, not character strings; comments & whitespace are *not* tokens (not true of all languages, cf. 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**
  - ifStmt
    - >=
      - ID(x)
      - ID(y)
    - assign
      - ID(y)
      - INT(42)
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
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 & function units
    - Memory hierarchy
Back End Structure

- Typically split into two major parts
  - “Optimization” – code improvements
    - Usually works on lower-level IR than AST
  - Code generation
    - Instruction selection & scheduling
    - Register allocation
The Result

- **Input**
  
  \[
  \text{if } (x \geq y) \\
  y = 42;
  \]

- **Output**
  
  \[
  \text{mov eax,}\left[\text{ebp}+16\right] \\
  \text{cmp eax,}\left[\text{ebp}-8\right] \\
  \text{jl L17} \\
  \text{mov } [\text{ebp}-8],42 \\
  \]

L17:
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)
  - 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, Itanium, 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”
  - 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
So what will this course cover?
- Only about 15% of you said “yes” for having had a compiler course, and what was covered was mixed, so...
- we will cover the basics, but quickly, then...
- we’ll explore more advanced things.
- If you are in that 15%, enjoy the review – but I’m guessing that everyone will pick up some new things.
CSE P 501 Course Project

- Best way to learn about compilers is to build (at least parts of) one
- Course project
  - Mini Java compiler: classes, objects, inheritance, etc.
  - 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 Details

- Goal: large enough language to be interesting; small enough to be tractable
- Project due in phases
  - Final result is the main thing, but timeliness and quality of intermediate work counts for something
  - Final report & short meeting at end of the course
- Core requirements, then open-ended
- Reasonably open to alternatives; let’s discuss
  - Have had people implement the compiler in C#, F# in the past; Haskell, ML, other languages with lex/yacc-like tools would make sense also
  - Tools, etc. can’t be proprietary – we need copies to run your code!

10/4/2011
Project Groups

- You are encouraged to work in groups of 2 (or maybe 3)
  - Suggestion: use class discussion board to find partners
- Space for SVN or other repositories + other shared files available on UW CSE machines
  - Use if desired; not required
  - Please send mail to perkins@cs with your and your partner’s CSE login ids if you want this
Programming Environments

- Whatever you want!
  - But if you’re using Java, your code should compile & run using standard Sun Oracle javac/java
  - If you use C# or something else, you assume some risk of the unknown
    - We’ll provide what pointers we can, but...
    - Work with other members of the class on infrastructure
    - Class discussion list can be very helpful here
  - If you’re looking for a Java IDE, try Eclipse
    - Or netbeans, or <name your favorite>
    - javac/java + emacs for the truly hardcore
Prerequisites

- Assume undergrad courses in:
  - Data structures & algorithms
    - Linked lists, dictionaries, trees, hash tables, graphs, &c
  - Formal languages & automata
    - Regular expressions, finite automata, context-free grammars, maybe a little parsing
  - Machine organization
    - Assembly-level programming for some machine (not necessarily x86)
- Gaps can usually be filled in
  - But be prepared to put in extra time if needed
Requirements & Grading

- Roughly
  - 50% project
  - 20% individual written homework
  - 25% exam (scheduled Thursday evening after Thanksgiving, Dec. 1, 6:30-8:00)
  - 5% other

- Homework submission online with feedback via the dropbox or email
CSE P 501 Administrivia

- 1 lecture per week
  - Tuesday 6:30-9:20, CSE 305 + MSFT
  - Carpools?
- Office Hours
  - Perkins: after class, drop-ins, CSE 548
  - Vasisht: Tue. 5:30-6:20, location tba
  - Also appointments
  - Suggestions for other times/locations?
Everything is (or will be) at
  www.cs.washington.edu/csep501
Lecture slides will be on the course web by mid-afternoon before each class
  Printed copies available in class at UW, but you may want to read or print in advance
Live video during class
  But do try to join us (questions, etc.) – it’s lonely talking to an empty room! (& not as good for you)
 Archived video and slides from class will be posted a day or two later
Communications

- Course web site
- Mailing list
  - You are automatically subscribed if you are enrolled
    - Your UW netid email – forward if needed
  - Will try to keep this fairly low-volume; limited to things that everyone needs to read
  - Link is on course web page
- Discussion board
  - Also linked from course web
  - Use for anything relevant to the course – let’s try to build a community
  - Can configure to have postings sent via email
Books

- Four good books (all on engr lib Reserve):
  - Aho, Lam, Sethi, Ullman, “Dragon Book”, 2nd ed (but 1st ed is also fine)
  - Cooper & Torczon, *Engineering a Compiler*
  - Fisher, Cryton, LeBlanc, *Crafting a Compiler*

- Cooper/Torczon book is the “official” text, but all would work & we’ll draw on all (and more). Older editions are generally okay.
Academic Integrity

- We want a cooperative group working together to do great stuff!
  - Possibilities include bounties for first person to solve vexing problems
- But: you must never misrepresent work done by someone else as your own, without proper credit
  - OK to share ideas & help each other out, but your project should ultimately be created by your group & solo homework / test should be your own
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

- Review of formal grammars
- Lexical analysis – scanning
  - Background for first part of the project
- Followed by parsing ...

- Good time to read the first couple of chapters of (any of) the book(s)