CSEP 501 – Compilers

Overview and Administrivia

Hal Perkins

Winter 2008
Credits

- Some direct ancestors of this course
  - Cornell CS 412-3 (Teitelbaum, Perkins)
  - Rice CS 412 (Cooper, Kennedy, Torczon)
  - UW CSE 401 (Chambers, Ruzzo, et al)
  - UW CSE 582 (Perkins)
  - Many grad compiler courses
  - Many books (Appel; Cooper/Torczon; Aho, [Lam,] Sethi, Ullman [Dragon Book])
Agenda

- Introductions
- What’s a compiler?
- Administrivia
CSEP 501 Personel

- **Instructor: Hal Perkins**
  - CSE 548; perkins@cs
  - Office hours: after class + drop in when you’re around + appointments

- **TA: Hao Lu**
  - hlv@cs
  - Office hours: CSE 216, Tue 5-6:15 + appointments
And the point is...

- Execute this!

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

- How?
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) 
    nPos++;
}
```
Interpreter

- Interpreter
  - Execution engine
  - Program execution interleaved with analysis
    
    ```
    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

- Well-known 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

- Variation: .NET
  - Compilers generate MSIL
  - All IL compiled to native code before execution
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)
  - Database engines, query languages
  - AI: 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’ll almost certainly write parsers and interpreters in some context if you haven’t already
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
Implications

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

- 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)
Tokens

- Token stream: Each significant lexical chunk of the program is represented by a token
  - Operators & Punctuation: { } [ ] ! + - = * ; : ... 
  - Keywords: if while return goto 
  - Identifiers: id & actual name 
  - Constants: kind & value; int, floating-point character, string, ...
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)
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
    - >=
    - assign
      - ID(x)
      - ID(y)
      - 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 consumption (pick some)
  - Should use machine resources effectively
    - Registers
    - Instructions
    - Memory hierarchy
Back End Structure

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

- **Input**
  
  ```
  if (x \geq y) 
  y = 42;
  ```

- **Output**
  
  ```
  mov eax,[ebp+16]
  cmp eax,[ebp-8]
  jl L17
  mov [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; Smalltalk & object-oriented)
  - New architectures (RISC machines, parallel machines, memory hierarchy issues)
  - More attention to back-end issues
Some History (3)

- 1990s and beyond
  - Compilation techniques appearing in many new places
    - Just-in-time compilers (JITs)
    - Software analysis, verification, security
  - Phased compilation – blurring the lines between “compile time” and “runtime”
    - Using machine learning techniques to control optimizations(!)
  - Compiler technology critical to effective use of new hardware (RISC, Itanium, complex memory hierarchies)
  - The new 800 lb gorilla - multicore
CSEP 501 Course Project

- Best way to learn about compilers is to build one
- CSEP 501 course project: Implement an x86 compiler in Java for an object-oriented programming language
  - MiniJava subset of Java from Appel book
  - Includes core object-oriented parts (classes, instances, and methods, including subclasses and inheritance)
  - Basic control structures (if, while)
  - Integer variables and expressions
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 conference at end of the course
- Core requirements, then open-ended
- Reasonably open to alternative projects; let’s discuss
  - Most likely would be a different implementation language (C#, ML, F#, ?) or target (MIPS/SPIM, x86-64, ...
Prerequisites

- Assume undergrad courses in:
  - Data structures & algorithms
    - Linked lists, dictionaries, trees, hash tables, &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
Project Groups

- You are encouraged to work in groups of 2 or 3
  - Pair programming strongly encouraged
- Space for CVS or SVN repositories + other shared files available on UW CSE machines
  - Use if desired; not required
  - Mail to instructor/TA if you want this
Programming Environments

- Whatever you want!
  - But assuming you’re using Java, your code should compile & run using standard Sun javac/java
    - Generics (Java 5/6) are fine
  - If you use C# or something else, you assume some risk of the unknown
    - Work with other members of the class and pull together
    - 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
Requirements & Grading

- Roughly
  - 50% project
  - 20% individual written homework
  - 25% exam (Thur. evening, about 2/3 of the way through the course)
  - 5% other

- Intent is to have homework submission online with graded work returned via email
  - Will adjust as needed
CSE 582 Administrivia

- 1 lecture per week
  - Tuesday 6:30-9:20, CSE 305 + MSFT
  - Carpoools?

- Office Hours
  - Perkins: after class, drop-ins
  - Lu: Tue. 5-6:15, UW CSE 216
  - Also appointments
  - Suggestions for other times/locations?
CSEP 501 Web

- 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.)
- 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
  - Will 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
Three good books:

- Aho, Lam, Sethi, Ullman, “Dragon Book”, 2\textsuperscript{nd} ed (but 1\textsuperscript{st} ed is also fine)
- Appel, \textit{Modern Compiler Implementation in Java}, 2\textsuperscript{nd} ed.
- Cooper & Torczon, \textit{Engineering a Compiler}

Dragon book is the “official” text, but all would work & we’ll draw on all three (and more)

If we put these on reserve in the engineering library, would anyone notice?
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 / tests 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)