


CSE 582 – Compilers

Overview and Administrivia
Hal Perkins
Autumn 2002


9/30/2002 © 2002 Hal Perkins & UW CSE A-1



Credits

- Some ancestors of this fall's CSE 582
 - Cornell CS 412-3 (Teitelbaum, Perkins)
 - Rice CS 412 (Cooper, Kennedy, Torczon)
 - UW CSE 401 (Chambers, Ruzzo, et al)
 - UW CSE 582 (Perkins)
 - Many books (particularly Cooper/Torczon; Aho, Sethi, Ullman [Dragon Book], Appel)


9/30/2002 © 2002 Hal Perkins & UW CSE A-2



Agenda for Today

- Introductions
- What's a compiler?
- CSE582 Administrivia


9/30/2002 © 2002 Hal Perkins & UW CSE A-3



CSE 582 Personnel

- n Instructor: Hal Perkins
 - n Sieg 208; perkins@cs
 - n Office hours: after class + drop whenever you're around and you can find me
- n TA: Nan Li
 - n annli@cs
 - n Office hours, etc. tbd

9/30/2002 © 2002 Hal Perkins & UW CSE A-4




And the point is...

- n Execute this!

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


9/30/2002 © 2002 Hal Perkins & UW CSE A-5



Interpreters & Compilers

- n Interpreter
 - n A program that reads an source program and produces the results of executing that program
- n Compiler
 - n A program that translates a program from one language (the *source*) to another (the *target*)

9/30/2002 © 2002 Hal Perkins & UW CSE A-6



Common Issues


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

```

while (k < length) { <nl> <tab> if (a[k] > 0
) <nl> <tab> <tab> { n P o s + + ; } <nl> <tab> }

```

9/30/2002 © 2002 Hal Perkins & UW CSE A-7



Interpreter


- Interpreter
 - Execution engine
 - Program execution interleaved with analysis

```

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

```
 - May involve repeated analysis of some statements (loops, functions)


9/30/2002 © 2002 Hal Perkins & UW CSE A-8



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


9/30/2002 © 2002 Hal Perkins & UW CSE A-9



Typical Implementations

- n Compilers
 - n FORTRAN, C, C++, Java, COBOL, etc. etc.
 - n Strong need for optimization, etc.
- n Interpreters
 - n PERL, Python, awk, sed, sh, csh, postscript printer, Java VM
 - n Effective if interpreter overhead is low relative to execution cost of language statements


9/30/2002 © 2002 Hal Perkins & UW CSE A-10



Hybrid approaches

- n Well-known example: Java
 - n Compile Java source to byte codes – Java Virtual Machine language (.class files)
 - n Execution
 - n Interpret byte codes directly, or
 - n Compile some or all byte codes to native code (particularly for execution hot spots)
 - n Just-In-Time compiler (JIT)
- n Variation: VS.NET
 - n Compilers generate MSIL
 - n All IL compiled to native code before execution


9/30/2002 © 2002 Hal Perkins & UW CSE A-11



Why Study Compilers? (1)

- n Become a better programmer(!)
 - n Insight into interaction between languages, compilers, and hardware
 - n Understanding of implementation techniques
 - n What is all that stuff in the debugger anyway?
 - n Better intuition about what your code does


9/30/2002 © 2002 Hal Perkins & UW CSE A-12



Why Study Compilers? (2)

- Compiler techniques are everywhere
 - Parsing (little languages, interpreters)
 - Database engines
 - AI: domain-specific languages
 - Text processing
 - Tex/LaTeX -> dvi -> Postscript -> pdf
 - Hardware: VHDL; model-checking tools
 - Mathematics (Mathematica, Matlab)


9/30/2002 © 2002 Hal Perkins & UW CSE A-13



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 solutions

9/30/2002 © 2002 Hal Perkins & UW CSE A-14



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 & hierarchy management, instruction set use

9/30/2002 © 2002 Hal Perkins & UW CSE A-15

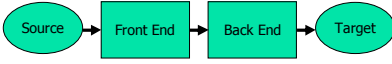
Why Study Compilers? (5)

- n You might even write a compiler some day!
 - n You'll almost certainly write parsers and interpreters if you haven't already

9/30/2002 © 2002 Hal Perkins & UW CSE A-16

Structure of a Compiler

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



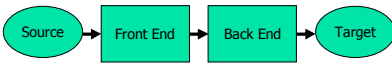
```

graph LR
  Source([Source]) --> FrontEnd[Front End]
  FrontEnd --> BackEnd[Back End]
  BackEnd --> Target([Target])
  
```

9/30/2002 © 2002 Hal Perkins & UW CSE A-17

Implications

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



```

graph LR
  Source([Source]) --> FrontEnd[Front End]
  FrontEnd --> BackEnd[Back End]
  BackEnd --> Target([Target])
  
```

9/30/2002 © 2002 Hal Perkins & UW CSE A-18

More Implications

- Need some sort of Intermediate Representation (IR)
- Front end maps source into IR
- Back end maps IR to target machine code

```

graph LR
    Source([Source]) --> FrontEnd[Front End]
    FrontEnd --> BackEnd[Back End]
    BackEnd --> Target([Target])
  
```

9/30/2002 © 2002 Hal Perkins & UW CSE A-19

Front End

```

graph LR
    source --> Scanner[Scanner]
    Scanner -- tokens --> Parser[Parser]
    Parser -- IR --> IR
  
```

- 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)

9/30/2002 © 2002 Hal Perkins & UW CSE A-20

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, ...

9/30/2002 © 2002 Hal Perkins & UW CSE A-21

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
- Note: tokens are atomic items, not character strings

9/30/2002 © 2002 Hal Perkins & UW CSE A-22

Parser Output (IR)

- Many different forms
 - (Engineering tradeoffs)
- Common output from a parser is an abstract syntax tree
 - Essential meaning of the program without the syntactic noise

9/30/2002 © 2002 Hal Perkins & UW CSE A-23

Parser Example

- Token Stream Input


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

- Abstract Syntax Tree


```

          graph TD
            ifStmt([ifStmt]) --> geq([>=])
            ifStmt --> assign([assign])
            geq --> idX([ID(x)])
            geq --> idY1([ID(y)])
            assign --> idY2([ID(y)])
            assign --> int42([INT(42)])
          
```


9/30/2002 © 2002 Hal Perkins & UW CSE A-24



Static Semantic Analysis

- During or (more common) after parsing
 - Type checking
 - Check for language requirements like "declare before use", type compatibility
 - Preliminary resource allocation
 - Collect other information needed by back end analysis and code generation


9/30/2002 © 2002 Hal Perkins & UW CSE A-25



Back End

- Responsibilities
 - Translate IR into target machine code
 - Should produce fast, compact code
 - Should use machine resources effectively
 - Registers
 - Instructions
 - Memory hierarchy

9/30/2002 © 2002 Hal Perkins & UW CSE A-26



Back End Structure

- Typically split into two major parts with sub phases
 - "Optimization" – code improvements
 - May well translate parser IR into another IR
 - We probably won't have time to do much with this part of the compiler
 - Code generation
 - Instruction selection & scheduling
 - Register allocation

9/30/2002 © 2002 Hal Perkins & UW CSE A-27

The Result

<ul style="list-style-type: none"> n Input <pre> if (x >= y) y = 42; </pre>	<ul style="list-style-type: none"> n Output <pre> mov eax,[ebp+16] cmp eax,[ebp-8] jl L17 mov [ebp-8],42 L17: </pre>
---	--

9/30/2002
© 2002 Hal Perkins & UW CSE
A-28

Some History (1)


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

9/30/2002
© 2002 Hal Perkins & UW CSE
A-29

Some History (2)

- n 1970's
 - n Syntax: formal methods for producing compiler front-ends; many theorems
- n 1980's
 - n New languages (functional; Smalltalk & object-oriented)
 - n New architectures (RISC machines, parallel machines, memory hierarchy issues)
 - n More attention to back-end issues

9/30/2002
© 2002 Hal Perkins & UW CSE
A-30




Some History (3)

- 1990's – now
 - Compilation techniques appearing in many new places
 - Just-in-time compilers (JITs)
 - Whole program analysis
 - Phased compilation – blurring the lines between “compile time” and “runtime”
 - Compiler technology critical to effective use of new hardware (RISC, Itanium, complex memories)

“May you study compilers in interesting times...”
Cooper & Torczon


9/30/2002 © 2002 Hal Perkins & UW CSE A-31



CSE 582 Course Project

- Best way to learn about compilers is to build one
- CSE 582 course project: Implement an x86 compiler in Java for an object-oriented programming language
 - Subset of Java
 - Includes core object-oriented parts (classes, instances, and methods, including inheritance)
 - Basic control structures (if, while)
 - Integer variables and expressions


9/30/2002 © 2002 Hal Perkins & UW CSE A-32



Project Details

- Goal: large enough language to be interesting; small enough to be tractable
 - With luck, get to some interesting back-end issues
- Project due in phases
 - Final result is the main thing, but timeliness and quality of intermediate work counts for something
 - Final report & conference at end of the course
- Core requirements, then open-ended
 - Core requirements: define what's needed to get a decent grade in the course
- Somewhat open to alternative projects; let's discuss
 - Most likely would be a different implementation language


9/30/2002 © 2002 Hal Perkins & UW CSE A-33



Prerequisites

- n Assume undergrad courses in:
 - n Data structures & algorithms
 - n Linked lists, dictionaries, trees, hash tables, &c
 - n Formal languages & automata
 - n Regular expressions, finite automata, context-free grammars, maybe a little parsing
 - n Machine organization
 - n Assembly-level programming for some machine (not necessarily x86)
- n Gaps can usually be filled in
 - n We'll review what we need when we get to it


9/30/2002 © 2002 Hal Perkins & UW CSE A-34



Project Groups

- n Students encouraged to work in groups of 2 (or maybe 3)
 - n Pair programming strongly encouraged
- n CVS repositories will be available on the UW CSE web
 - n Use if desired; not required


9/30/2002 © 2002 Hal Perkins & UW CSE A-35



Programming Environments

- n Whatever you want!
 - n But assuming you're using Java, your code should compile & run with the standard Sun javac/java tools (best to use Java 1.2 or later)
 - n If you use C# or something else, you assume the risk of the unknown
 - n Work with other members of the class and pull together
- n We'll put links to various tools on our web
 - n Many (most?) are free downloads


9/30/2002 © 2002 Hal Perkins & UW CSE A-36



Requirements & Grading

- n Roughly
 - n 50% project
 - n 20% written homework
 - n 25% single (midterm) exam
 - n 5% other
- n All homework submission will be online
 - n Graded work will be returned via email


9/30/2002 © 2002 Hal Perkins & UW CSE A-37



CSE 582 Administrivia

- n 2 lectures per week
 - n T, Th 6:30-7:50, Sieg 322 or at Microsoft
 - n Feel free to switch back & forth as desired
 - n Do we want/need a break in the middle?
 - n Carpools?
- n Office Hours
 - n Perkins: after class
 - n Li: tbd (preferences?)


9/30/2002 © 2002 Hal Perkins & UW CSE A-38



Java Boot Camp

- n If the demand is there, we could run a Java boot camp
 - n 2-3 hour lecture about Java language basics
 - n Probably Saturday afternoon, but could do it as a long lecture this Thursday
 - n How much interest?


9/30/2002 © 2002 Hal Perkins & UW CSE A-39



CSE 582 Web

- n Everything is at www.cs.washington.edu/582
- n Lecture slides will be available by mid-afternoon before each class
 - n Will be linked from the top-level web page
 - n Printed copies available in UW classroom; if you are a distance student, we strongly suggest you print a copy in advance for notes, etc.


9/30/2002 © 2002 Hal Perkins & UW CSE A-40



Communications

- n Course web site
- n Mailing list
 - n You will be automatically subscribed if you are enrolled
 - n Want this to be fairly low-volume; limited to things that everyone needs to read
 - n Link will appear on course web page
- n Discussion board
 - n Also linked from course web
 - n Use for anything relevant to the course – let's try to build a community
- n IM? Online office hours? Other ideas?


9/30/2002 © 2002 Hal Perkins & UW CSE A-41



Books

- n Main textbook: *Engineering a Compiler* by Keith Cooper & Linda Torczon
 - n Not yet available in bookstores
 - n Preprints available at Professional Copy & Print, Univ. Way & 42nd St. (aprox \$40, tax incl.)
- n A couple of other good compiler books
 - n Aho, Sethi, Ullman, "Dragon Book"
 - n Appel, "Modern Compiler Implementation in Java"
 - n If we put these on reserve in the engineering library, would anyone notice?


9/30/2002 © 2002 Hal Perkins & UW CSE A-42



Academic Integrity

- n Goal: create a cooperative community working together to learn and implement great projects!
 - n Possibilities include bounties for first person to solve vexing problems
- n But: you must never misrepresent work done by someone else as your own, without proper credit
 - n OK to share ideas & help each other out, but your project should ultimately be created by your group


9/30/2002 © 2002 Hal Perkins & UW CSE A-43



Any questions?

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

9/30/2002 © 2002 Hal Perkins & UW CSE A-44



Coming Attractions

- n Review of formal grammars
- n Lexical analysis – scanning
 - n First part of the project
- n Followed by parsing...

- n Suggestion: read the first couple of chapters of the book

9/30/2002 © 2002 Hal Perkins & UW CSE A-45
