


## CSEP 501 – Compilers

---

Overview and Administrivia  
Hal Perkins  
Autumn 2005


10/18/2005 © 2002-5 Hal Perkins & UW CSE A-1



## Credits

- Some ancestors of this summer's 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 books (particularly Cooper/Torczon; Aho, Sethi, Ullman [Dragon Book], Appel)


10/18/2005 © 2002-5 Hal Perkins & UW CSE A-2



## Agenda

- Introductions
- What's a compiler?
- Administrivia


10/18/2005 © 2002-5 Hal Perkins & UW CSE A-3



## CSEP 501 Personnel

- Instructor: Hal Perkins
  - CSE 548; perkins@cs
  - Office hours: after class + drop whenever you're around and you can find me + appointments
- TA: Yael Schwartzman
  - yaels@cs
  - Office hours: Tue. 4:30-6:30 pm, CSE 218 + appointments

10/18/2005 © 2002-5 Hal Perkins & UW CSE A-4




## And the point is...

- Execute this!

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

10/18/2005 © 2002-5 Hal Perkins & UW CSE A-5



## Interpreters & Compilers

- Interpreter
  - A program that reads a 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*)

10/18/2005 © 2002-5 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> }
```

10/18/2005

© 2002-5 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)

10/18/2005

© 2002-5 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

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-9

## Typical Implementations

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

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-10

## 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
- Variation: .NET
  - Compilers generate MSIL
  - All IL compiled to native code before execution

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-11

## 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

10/18/2005

© 2002-5 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)

10/18/2005 © 2002-5 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

10/18/2005 © 2002-5 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, instruction set use, memory hierarchy management

10/18/2005 © 2002-5 Hal Perkins & UW CSE A-15

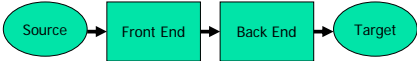
## Why Study Compilers? (5)

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

10/18/2005 © 2002-5 Hal Perkins & UW CSE A-16

## 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



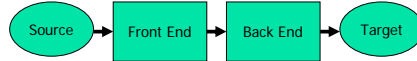
```

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

10/18/2005 © 2002-5 Hal Perkins & UW CSE A-17

## Implications

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



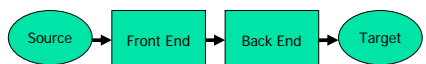
```

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

10/18/2005 © 2002-5 Hal Perkins & UW CSE A-18

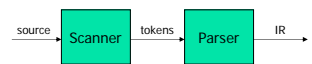
## More Implications

- Need some sort of Intermediate Representation(s) (IR)
- Front end maps source into IR
- Back end maps IR to target machine code
- May be multiple IRs – higher level at first, lower level in later phases



10/18/2005 © 2002-5 Hal Perkins & UW CSE A-19

## 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)

10/18/2005 © 2002-5 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, ...

10/18/2005 © 2002-5 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

  - Notes: tokens are atomic items, not character strings; comments are *not* tokens

10/18/2005 © 2002-5 Hal Perkins & UW CSE A-22

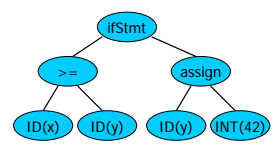
## Parser Output (IR)

- Many different forms
  - Engineering tradeoffs that have changed over time
- Common output from a parser is an abstract syntax tree
  - Essential meaning of the program without the syntactic noise

10/18/2005 © 2002-5 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
 

10/18/2005 © 2002-5 Hal Perkins & UW CSE A-24

## Static Semantic Analysis

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

10/18/2005

© 2002-5 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

10/18/2005

© 2002-5 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 other IRs
    - We probably won't have time to do much with this part of the compiler, alas
  - Code generation
    - Instruction selection & scheduling
    - Register allocation

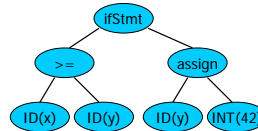
10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-27

## The Result

- Input
  - if (x >= y)
  - y = 42;
- Output
  - mov eax,[ebp+16]
  - cmp eax,[ebp-8]
  - jl L17
  - mov [ebp-8],42
  - L17:



10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-28

## 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.

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-29

## 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

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-30

## Some History (3)

- Since 1990
  - 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”
    - Using machine learning techniques to control optimizations(!)
  - Compiler technology critical to effective use of new hardware (RISC, Itanium, complex memory hierarchies)

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-31

## 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 with subclasses
  - Includes core object-oriented parts (classes, instances, and methods, including inheritance)
  - Basic control structures (if, while)
  - Integer variables and expressions

10/18/2005

© 2002-5 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: what's needed to get a decent grade in the course
- Reasonably open to alternative projects; let's discuss
  - Most likely would be a different implementation language (C#, ML?) or target (MIPS/SPIM, x86-64, ...)

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-33

## 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
  - We'll review what we need when we get to it

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-34

## Project Groups

- Students encouraged to work in groups of 2 or 3
  - Pair programming strongly encouraged
- Space for CVS repositories will be available from UW CSE
  - Use if desired; not required

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-35

## Programming Environments

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

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-36

## 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

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-37

## CSE 582 Administrivia

- 1 lecture per week
  - Tuesday 6:30-9:20, CSE 305
  - Carpools?
- Office Hours
  - Perkins: after class
  - Schwartzman: Tuesday 4:30-6:30
  - Also appointments

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-38

## CSEP 501 Web

- Everything is (or will be) at [www.cs.washington.edu/csep501](http://www.cs.washington.edu/csep501)
- Lecture slides will be available on the course web by mid-afternoon before each class
  - Printed copies available in class, but you may want to read or print in advance

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-39

## Communications

- Course web site
- Mailing list
  - You will be automatically subscribed if you are enrolled
  - Want this to be fairly low-volume; limited to things that everyone needs to read
  - Link will appear 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
- IM? Online office hours? Other ideas?

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-40

## Books

- Main textbook: Appel, *Modern Compiler Implementation in Java*, 2<sup>nd</sup> ed.
- A couple of other good compiler books
  - Aho, Sethi, Ullman, "Dragon Book"
  - Cooper & Torczon, *Engineering a Compiler*
    - If we put these on reserve in the engineering library, would anyone notice?

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-41

## Academic Integrity

- Goal: create a cooperative community working together to learn and implement great projects!
  - 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

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-42



## Any questions?

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

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-43



## Coming Attractions

- Review of formal grammars
- Lexical analysis – scanning
  - First part of the project
- Followed by parsing...
  
- Suggestion: read the first couple of chapters of the book

10/18/2005

© 2002-5 Hal Perkins & UW CSE

A-44