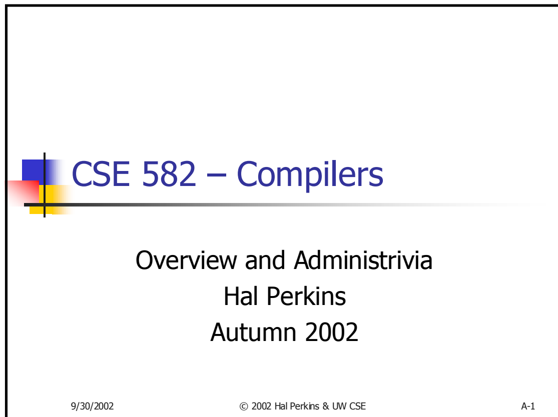
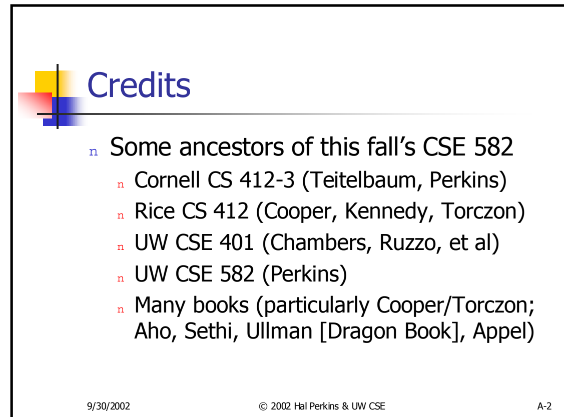
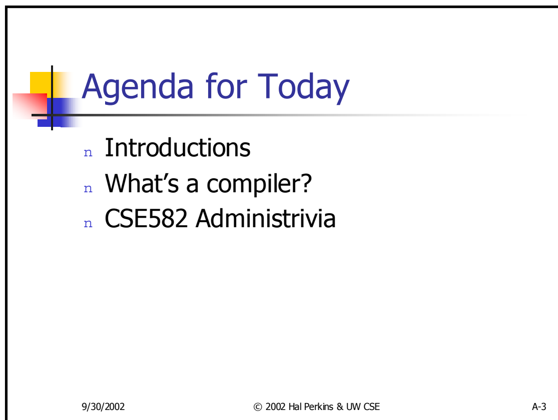
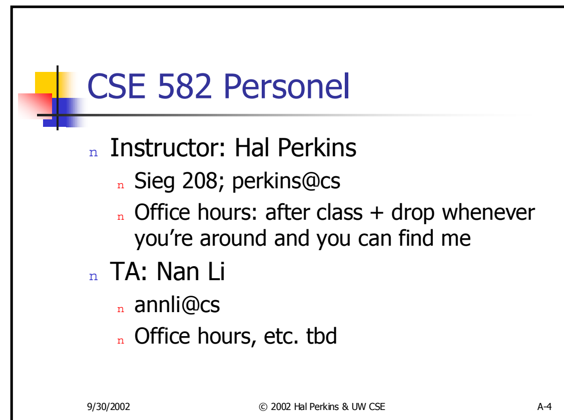
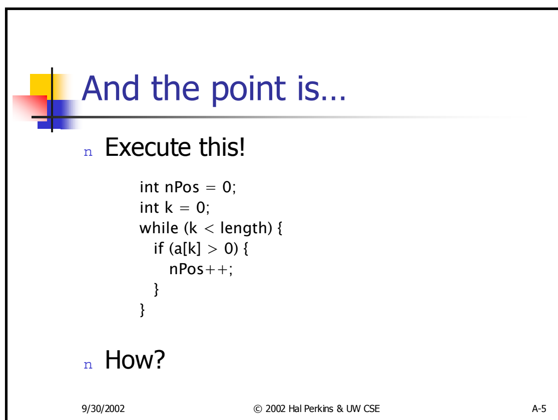
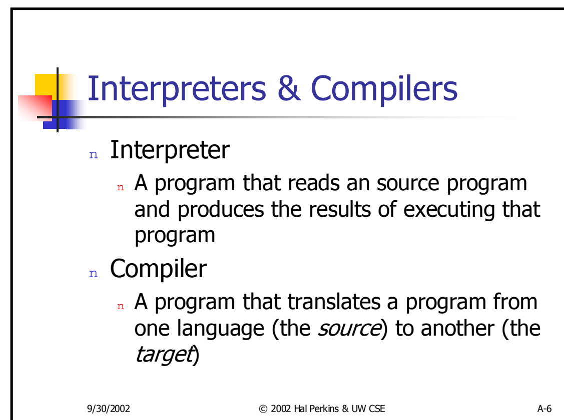

CSE 582 – Compilers
Overview and Administrivia
Hal Perkins
Autumn 2002
9/30/2002
© 2002 Hal Perkins & UW CSE
A-1


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 Personel
Instructor: Hal Perkins
Sieg 208; perkins@cs
Office hours: after class + drop whenever you're around and you can find me
TA: Nan Li
annli@cs
Office hours, etc. tbd
9/30/2002
© 2002 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?
9/30/2002
© 2002 Hal Perkins & UW CSE
A-5


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)
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*

```
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> }
```

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

- Compilers
  - FORTRAN, C, C++, Java, COBOL, etc. etc.
  - Strong need for optimization, etc.
- Interpreters
  - PERL, Python, awk, sed, sh, csh, postscript printer, Java VM
  - 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

- 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
      - (particularly for execution hot spots)
      - Just-In-Time compiler (JIT)
- Variation: VS.NET
  - Compilers generate MSIL
  - All IL compiled to native code before execution

9/30/2002 © 2002 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

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

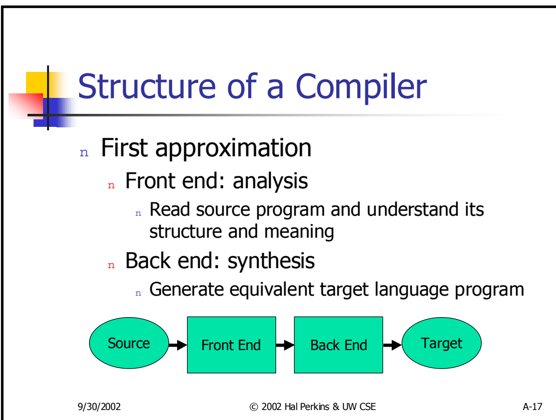
Why Study Compilers? (5)

- You might even write a compiler some day!
  - 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

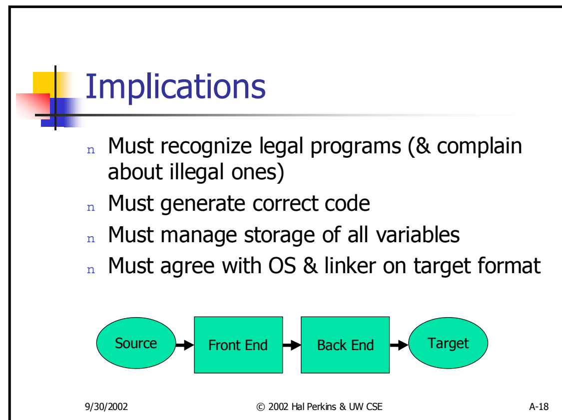
- First approximation
  - Front end: analysis
    - Read source program and understand its structure and meaning
  - Back end: synthesis
    - Generate equivalent target language program



9/30/2002 © 2002 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



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

Source → Front End → Back End → Target

 © 2002 Hal Perkins & UW CSE

Front End

source → Scanner → tokens → Parser → 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)

 © 2002 Hal Perkins & UW CSE

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, …

 © 2002 Hal Perkins & UW CSE

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

 © 2002 Hal Perkins & UW CSE

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

 © 2002 Hal Perkins & UW CSE

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)

 © 2002 Hal Perkins & UW CSE

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

Back End

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

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

The Result

- Input

```
if (x >= 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
  - Formal notations for syntax
  - Fundamental implementation techniques
    - Stack frames, recursive procedures, etc.

Some History (2)

- 1970's
  - Syntax: formal methods for producing compiler front-ends; many theorems
- 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)

- 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

- 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

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

Project Groups

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

9/30/2002 © 2002 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 (best to use Java 1.2 or later)
  - If you use C# or something else, you assume the risk of the unknown
    - Work with other members of the class and pull together
- We'll put links to various tools on our web
  - Many (most?) are free downloads

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

Requirements & Grading

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

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

CSE 582 Administrivia

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

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

Java Boot Camp

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

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

CSE 582 Web

- Everything is at www.cs.washington.edu/582
- Lecture slides will be available by mid-afternoon before each class
  - Will be linked from the top-level web page
  - 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

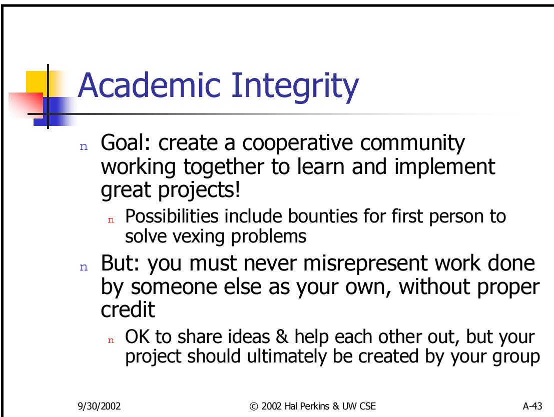
- Course web site
- Mailing list
  - You will be automatically subscribed if you are enrolled
  - Want this to be fairly low-volume; limited to things that everyone needs to read
  - Link will appear on course web page
- Discussion board
  - Also linked from course web
  - Use for anything relevant to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

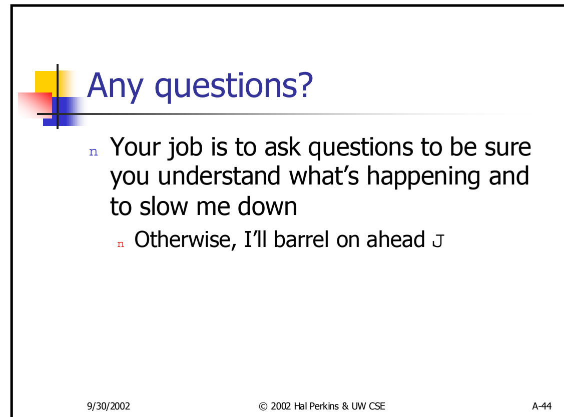
9/30/2002 © 2002 Hal Perkins & UW CSE A-41

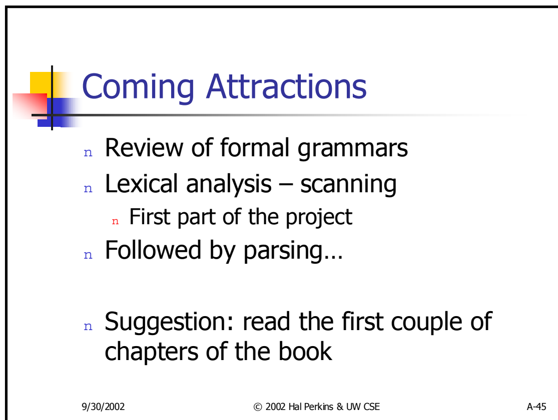
Books

- Main textbook: *Engineering a Compiler* by Keith Cooper & Linda Torczon
  - Not yet available in bookstores
  - Preprints available at Professional Copy & Print, Univ. Way & 42nd St. (aprox $40, tax incl.)
- A couple of other good compiler books
  - Aho, Sethi, Ullman, "Dragon Book"
  - Appel, "Modern Compiler Implementation in Java"
    - 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
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
9/30/2002
© 2002 Hal Perkins & UW CSE
A-43


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


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