Credits

- Most of the material in the following lectures is derived from lectures taught by Hal Perkins for CSE 413 and CSE 582
- He also credits previous classes ...  
  » Cornell CS 412-3 (Teitelbaum, Perkins)
  » Rice CS 412 (Cooper, Kennedy, Torczon)
  » UW CSE 401 (Chambers, Ruzzo, et al)
  » Many books (particularly Cooper/Torczon; Aho, Sethi, Ullman [Dragon Book], Appel)

Books

- *Engineering a Compiler* by Keith Cooper & Linda Torczon
  » Not yet available in bookstores
  » Preprints available at Professional Copy & Print, Univ. Way & 42nd St. (~ $40, tax incl.)
  » the “Dragon Book”
- *Modern Compiler Implementation in Java*, by Appel

Why are we doing this?

- Execute this!
  ```
  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

```plaintext
while (k < length)
  if (a[k] > 0)
    { pos++; }
```

Interpreter

- Interpreter
  » Execution engine
  » Program execution interleaved with analysis
    ```plaintext
    running = true;
    while (running) {
      analyze next statement;
      execute that statement;
    }
    ```
  » May involve repeated analysis of some statements (loops, functions)

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, C#, 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
  » Functional languages like Scheme and Smalltalk where the environment is dynamic

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

Why Study Compilers? Programmer

• 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? Designer

• 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)
Why Study Compilers? *Theoretician*

- 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

Why Study Compilers? *Education*

- 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

- You might even write a compiler some day!
  - You’ll almost certainly write parsers and interpreters 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
- Must agree with OS & linker on target format
More Implications

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

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 or by hand
  » 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

  » Note: tokens are atomic items, not character strings
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

Parser Example

- Token Stream Input
- Abstract Syntax Tree

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 subphases
  - “Optimization” – code improvements
    - May well translate parser IR into another IR
    - We won’t 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 Ancient History

- 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 Later History

- 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 Recent History

- 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