Compiler Design and Construction

CSE 4102

Overview

Acknowledgement

- Hal Perkins, University of Washington
- Amin Ahsan Ali, Assistant Professor and Iffat Anjum, Lecturer, Department of Computer Science and Engineering, University of Dhaka

Outline

- Introductions
- What's a compiler?
- Administrivia

Course Instructor

Mubin Ul Haque (MUH)

Lecturer, Dept. of CSE, University of Dhaka

- Email: mubin10haque@gmail.com
- Please use ‘CSE 4102’ as subject
- Phone: 01671 73 37 13

- Class hours:

  Monday (10.00 am -11.30 am)

  Sunday (11:30 am – 1:00 pm)

- Personal Site: https://sites.google.com/view/mubinulhaque/home
- Google Classroom Code: pvdwszl

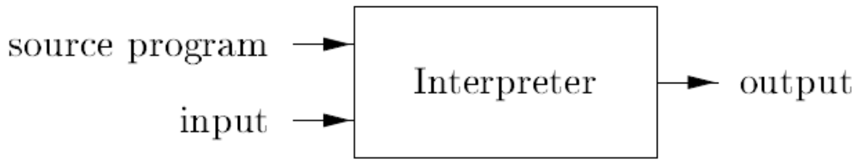
- Execute this!

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

And the point is…

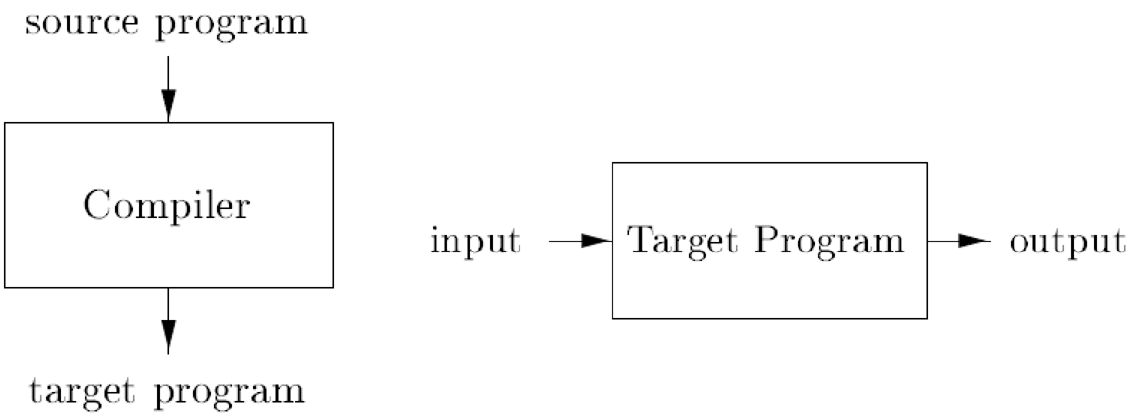
- How?

- Interpreter
  - A program that reads an source program and produces the results of executing that program



Interpreters & Compilers

- 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*

```
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> }
```

- **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

Interpreter

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 (Microsoft Intermediate Language)
  - All IL compiled to native code before execution

Why Study Compilers?

- 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?

- 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?

- 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?

- 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

- You might even write a compiler some day!

  - You’ll almost certainly write parsers and interpreters in some context if you haven’t already

Some History

- 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

- 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

- 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

Books

1. [Required] A Aho, M Lam, R Sethi, J Ullman, **Compilers - Principles, Techniques, and Tools**, 2nd edition, Addison Wesley.
2. K Cooper and L Torczon, **Engineering a Compiler**, 2nd edition, Morgan Kaufmann Steven Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann Publishers, 1997.
3. [Required] J Levine, T Mason, D Brown, Lex and Yacc, 1st edition, O'Reilly [search in google books] or J Levine, Flex and Bison, **O'Reilly Computer Architecture : A Quantitative Approach** (Appendix A - Assemblers, Linkers, and the SPIM Simulator) http://pages.cs.wisc.edu/~larus/HP_AppA.pdf

Prerequisites

- 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

Grading Policy

- Roughly
  - 60% - Final Examination
  - 30% - In course Examination (No additional in-course exam will be considered)
  - 5% - One/Two Surprise Quiz (subject to change)
  - 5% - Attendance (subject to change)

Surprise Quizzes

- There will be surprise quizzes, given at the start of a lecture, or during any lecture.
- NO LATE or MAKEUP SURPRISE QUIZZES, under any circumstances whatsoever.
- Surprise quizzes are completely individual efforts.

Playing it safe

If you follow these 4 simple rules during the class, you'll make sure that you do well in the course:

1. Attend every Theory and LAB classes.

2. Read the course material (textbook sections assigned + slides).

3. Submit everything (Assignments, Quizzes, Exams) on time - don't be late.

4. Don't cheat.

Course Outline (tentative)

Topic		Lectures
Introductory Class		1
Compiler Overview		1
Lexical Analysis: Tokens, Recognition of tokens: Finite Automata, Regular Expressions, LEX		2
Syntax Analysis		6
	Context Free Grammars and Recursive Descent Parsing	1
	Top-down Parsing: LL(1) parsers	2
	Bottom-up Parsing: LR(1) parsers	2
	YACC and Error Handling	1
Semantic Analysis & Syntax Directed Translation		2
Discussion and problem solving		1
In-course Exam		

Course Outline (tentative)

Topic	Lectures
Intermediate Code Generation: 3-address codes, Static Single Assignment (SSA) Forms, Translation Schemes for Expressions, Array Reference and Control Flow Statements	3
Run Time Environment: Stack Allocation, Activation Records, Heap Management, Garbage Collection	2
Code Generation	3
Control flow Graphs and Peephole Optimization	1
Simple Code Generator	1
Global Register Allocation	1
Optimization	4
Data Flow Analysis - Reaching Definitions, Live-variable Analysis, Available Analysis, Partial Redundancy Elimination, Loop Optimization	2
Code Scheduling - Data and Control Dependency, Basic Block Scheduling and Global Code Scheduling	2
Discussion and problem solving	1
Total Number of Class	**26**

Question?