Why Take Programming Languages and Compilers?
Why Take Programming Languages and Compilers?

To appreciate the marriage of theory and practice

“Theory and practice are not mutually exclusive; they are intimately connected. They live together and support each other.”

[D.E. Knuth, 1989]
Theory in Practice: Regular Expression Pattern Matching in Perl, Python, Ruby vs. AWK

Time to check whether $a^n a^n$ matches $a^n$

Russ Cox, Regular expression matching can be simple and fast (but is slow in Java, Perl, PHP, Python, Ruby, ...) [http://swtch.com/~rsc/regexp/regexp1.html, 2007]
Why Take Programming Languages and Compilers?

To appreciate the marriage of theory and practice

To explore the dimensions of computational thinking

To exercise creativity

To learn robust software development practices
What is a Programming Language?

A programming language is a notation for describing computations to people and to machines.
Computational Thinking in Programming Language Design

Underlying every programming language is a **model of computation**:

- **Procedural**: C, C++, C#, Java
- **Declarative**: SQL
- **Logic**: Prolog
- **Functional**: Haskell
- **Scripting**: AWK, Perl, Python, Ruby
Evolutionary Forces on Languages and Compilers

More and different kinds of languages
Increasing diversity of applications
Stress on increasing productivity
Need to improve software reliability
Target machines more diverse
Parallel machine architectures
Massive compiler collections
Evolution of Programming Languages: 1970 to 2010

<table>
<thead>
<tr>
<th>1970</th>
<th>2010</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>Java</td>
</tr>
<tr>
<td>Lisp</td>
<td>C</td>
</tr>
<tr>
<td>Cobol</td>
<td>PHP</td>
</tr>
<tr>
<td>Algol 60</td>
<td>C++</td>
</tr>
<tr>
<td>APL</td>
<td>Visual Basic</td>
</tr>
<tr>
<td>Snobol 4</td>
<td>C#</td>
</tr>
<tr>
<td>Simula 67</td>
<td>Python</td>
</tr>
<tr>
<td>Basic</td>
<td>Perl</td>
</tr>
<tr>
<td>PL/1</td>
<td>Delphi</td>
</tr>
<tr>
<td>Pascal</td>
<td>JavaScript</td>
</tr>
</tbody>
</table>

[http://www.tiobe.com]
Today there are thousands of programming languages.

The website [http://www.99-bottles-of-beer.net](http://www.99-bottles-of-beer.net) has programs in 1,271 different programming languages to print the lyrics to the song “99 Bottles of Beer.”
“99 Bottles of Beer”

99 bottles of beer on the wall, 99 bottles of beer.
Take one down and pass it around, 98 bottles of beer on the wall.

98 bottles of beer on the wall, 98 bottles of beer.
Take one down and pass it around, 97 bottles of beer on the wall.

  
  
  
2 bottles of beer on the wall, 2 bottles of beer.
Take one down and pass it around, 1 bottle of beer on the wall.

1 bottle of beer on the wall, 1 bottle of beer.
Take one down and pass it around, no more bottles of beer on the wall.

No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, 99 bottles of beer on the wall.

[Traditional]
“99 Bottles of Beer” in AWK

BEGIN {
    for(i = 99; i >= 0; i--) {
        print ubottle(i), "on the wall," , lbottle(i) "."
        print action(i), lbottle(inext(i)), "on the wall."
        print
    }
}

function ubottle(n) {
    return sprintf("%s bottle%s of beer", n ? n : "No more", n - 1 ? "s" : ")
}

function lbottle(n) {
    return sprintf("%s bottle%s of beer", n ? n : "no more", n - 1 ? "s" : ")
}

function action(n) {
    return sprintf("%s", n ? "Take one down and pass it around," : "Go to the store and buy some more,"

}

function inext(n) {
    return n ? n - 1 : 99
}

"99 Bottles of Beer" in Perl

[Andrew Savage, http://search.cpan.org/dist/Acme-EyeDrops/lib/Acme/EyeDrops.pm]
“99 Bottles of Beer” in the Whitespace Language

[Edwin Brady and Chris Morris, U. Durham]
Conlangs: Made-Up Languages

Okrent lists 500 invented languages including:

• Lingua Ignota [Hildegard of Bingen, c. 1150]

• Esperanto [L. Zamenhof, 1887]

• Klingon [M. Okrand, 1984]
  Huq Us'pty G'm (I love you)

• Proto-Central Mountain [J. Burke, 2007]

• Dritok [D. Boozer, 2007]
  Language of the Drushek, long-tailed beings with large ears and no vocal cords

[Arika Okrent, In the Land of Invented Languages, 2009]
[http://www.inthelandofinventedlanguages.com]
What is a Compiler?

source program → Compiler → target program

input → output
Target Languages

Another programming language
CISCs
RISCs
Vector machines
Multicores
GPUs
Quantum computers
An Interpreter Directly Executes a Source Program on its Input
Java Compiler

source program

Translator

intermediate representation

input

Java Virtual Machine

output
Compilers Can Have Many Other Forms

- **Cross compiler**: a compiler on one machine that generates target code for another machine
- **Incremental compiler**: one that can compile a source program in increments
- **Just-in-time compiler**: one that is invoked at runtime to compile each called method in the IR to the native code of the target machine
- **Ahead-of-time compiler**: one that translates IR to native code prior to program execution
What Should We Teach?

• Unifying abstractions

• Fundamental models

• Basic algorithms

• Computational thinking
Regular expressions generate the regular sets
\[ a(a|b)^* \] generates all strings of \( a \)'s and \( b \)'s beginning with an \( a \)

Finite automata recognize the regular sets

This automaton recognizes the same set of strings.
This grammar $G$ generates all strings of $a$’s and $b$’s with the same number of $a$’s as $b$’s:

$$S \rightarrow aAbS | bBaS | \varepsilon$$
$$A \rightarrow aAbA | \varepsilon$$
$$B \rightarrow bBaB | \varepsilon$$

$G$ is unambiguous and has only one parse tree for every sentence in $L(G)$. 
There is an Art to Writing Good Grammars

The grammar $S \rightarrow aSbS \mid bSaS \mid \varepsilon$ also generates all strings of $a$’s and $b$’s with the same number of $a$’s as $b$’s.

But this grammar is ambiguous: $abab$ has two parse trees

$(ab)^n$ has $\frac{1}{n+1}\binom{2n}{n}$ parse trees
Natural Languages are Inherently Ambiguous

I made her duck.

[5 meanings: D. Jurafsky and J. Martin, 2000]

One morning I shot an elephant in my pajamas. How he got into my pajamas I don’t know.

[Groucho Marx, Animal Crackers, 1930]

List the sales of the products produced in 1973 with the products produced in 1972.

[455 parses: W. Martin, K. Church, R. Patil, 1987]
Methods for Specifying the Semantics of Programming Languages

Operational semantics

translation of program constructs to an understood language

Axiomatic semantics

assertions called preconditions and postconditions specify the properties of statements

Denotational semantics

semantic functions map syntactic objects to semantic values
Principles of Compiler Design circa 1977

- Introduction to compilers
- Programming languages
- Finite automata and lexical analysis
- Syntactic specification of programming languages
- Basic parsing techniques
- Automatic construction of efficient parsers
- Syntax-directed translation
- Symbol tables
- Run-time storage
- Error detection and recovery
- Code optimization
- Data-flow analysis
- Code generation
Phases of a Compiler

source program


token stream

syntax tree

annotated syntax tree

interm. rep.

interm. rep.

target program

Symbol Table
Compiler Component Generators

Lexical Analyzer Generator (lex)

Syntax Analyzer Generator (yacc)

Lexical Analyzer

Syntax Analyzer

source program

token stream

syntax tree
Lex Specification for a Desk Calculator

```
number        [0-9]+\.?[0-9]*\.[0-9]+  
  \%
[  ]        { /* skip blanks */ }
{number}     { sscanf(yytext, "%lf", &yylval);  
              return NUMBER; }  
\n|.        { return yytext[0]; }  
```
Yacc Specification for a Desk Calculator

%token NUMBER
%left '+'
%left '*'
%
lines : lines expr \n' { printf("%g\n", $2); } |
 | /* empty */
 |
expr : expr '+' expr   { $$ = $1 + $3; }
 | expr '*' expr   { $$ = $1 * $3; }
 | '(' expr ')'
 | NUMBER
 |
%
#include "lex.yy.c"
Creating the Desk Calculator

Invoke the commands

```
lex desk.l
yacc desk.y
cc y.tab.c -ly -ll
```

Result

```
1.2 * (3.4 + 5.6) → Desk Calculator → 10.8
```
Added Topics circa 2010

- Garbage collection
- Data-flow analysis schemas
- Instruction-level parallelism
- Optimizing for parallelism and locality
- Interprocedural analysis
- New intermediate representations
  - Static single-assignment form
  - MSIL
- New tools
  - ANTLR
  - Phoenix
- Compilers now come in collections
  - GCC
  - .NET
The Compilers Course at Columbia

• In PLT you will learn the syntactic and semantic elements of the most important modern programming languages as well as the algorithms and techniques used by compilers to translate them into machine and other target languages. The course will cover imperative, object-oriented, functional, logic, and scripting languages.

• A highlight of this course is a semester-long programming project in which you will work in a small team to create and implement an innovative little language of your own design. This project will teach you project management, teamwork, and communication skills that you can apply in all aspects of your career.
The Compilers Course Project at Columbia

**Week**  **Task**
2  **Form a team of five** and design an innovative new language
4  **Write a whitepaper** on your proposed language modeled after the Java whitepaper
8  **Write a tutorial** patterned after Chapter 1 and a **language reference manual** patterned after Appendix A of Kernighan and Ritchie’s book, *The C Programming Language*
14  **Give a ten-minute presentation** of the language to the class
15  **Give a 30-minute working demo** of the compiler to the teaching staff
15  **Hand in the final project report**
Team Roles

• **Project manager**
  – sets the project schedule, holds weekly meetings with the entire team, maintains the project log, and makes sure the project deliverables get done on time.

• **Language and tools guru**
  – defines the baseline process to track language changes and maintain the intellectual integrity of the language.
  – teaches the team how to use various tools used to build the compiler.

• **System architect**
  – defines the compiler architecture, modules, and interfaces.

• **System integrator**
  – defines the system platform and makes sure the compiler components work together.

• **Tester and validator**
  – defines the test suites and executes them to make sure the compiler meets the language specification.
Some of the Languages Created in the Compilers Course at Columbia in the Fall Semester 2009

BALL: simulating baseball games

Celluloid: interactive media sequencing

EZasPI: expressing zombies as programmable individuals

KAction: martial arts instruction

Pixel Power: image and graphics processing

Twinkle: creating interactive activities for toddlers
Pixel Power is designed to make it easier and more intuitive to write software based on the stream processing paradigm. It has built-in data types that are conducive to image and matrix operations. It uses Microsoft’s DirectX High Level Shader Language (HLSL) on Windows.
Pixel Power: Language Brainstorming

Kaushik Viswanathan

System architect of the
Pixel Power project

• Designed and implemented the graphics libraries, and the links between the translator modules and the compiler environment
Pixel Power: Project Management

Eric Liu

Project manager of the
Pixel Power project

• Organized team meetings, implemented compiler preprocessing and shared structures
Eric Liu: Project Management - 1

- A diverse team from different backgrounds
- Compromise: we don’t know each other, make everyone happy vs. focus
- Expertise: focus our efforts behind one individual’s extensive background
Eric Liu: Project Management - 2

- Communication: after every class, bulletin board discussion, shared meeting notes & documents

- Loss of team member: redistribute jobs, pare down grammar

- The best ways for getting along in a new team may not get things done. Push ahead.
Pixel Power: Lessons Learned

Eliane Kabkab

Systems integrator of the Pixel Power project

• Defined the work environment and tools

• Designed and implemented syntax translation into the target code
What is Q-HSK?

Q-HSK is a language which we designed to facilitate the implementation and simulation of quantum algorithms on a classical computer.
Computational Thinking for Designing Quantum Computer Programming Languages

- Physical System
- Mathematical Formulation
- Discretization
- Model of Computation
Quantum Computer Compiler

Computational abstractions

K. Svore, A. Aho, A. Cross, I. Chuang, I. Markov
A Layered Software Architecture for Quantum Computing Design Tools
QCC for Ion Trap Quantum Computing Device

Mathematical Model: Quantum mechanics, unitary operators, tensor products

Computational Formulation: Quantum bits, gates, and circuits

QIR to QASM

Target QPOL

Physical System: Laser pulses applied to ions in traps

EPR Pair Creation

Quantum Circuit Model

QIR

QASM

QPOL

Machine Instructions

Physical Device

\[ |0,y\rangle + \frac{(-1)^y |1,y\rangle}{\sqrt{2}}, \text{ for } x, y \in \{0, 1\} \]
Design Flow with Fault Tolerance and Error Correction

- **Mathematical Model:** Quantum mechanics, unitary operators, tensor products
- **Computational Formulation:** Quantum bits, gates, and circuits
- **QCC:** QIR, QASM
- **Software:** QPOL
- **Physical System:** Laser pulses applied to ions in traps

**Quantum Circuit Model**

- EPR Pair Creation
  \[ |x\rangle + (-1)^x |y\rangle \]

- **Fault Tolerance and Error Correction (QEC)**

- **Machine Instructions**
  
- **Physical Device**

- **K. Svore**
  PhD Thesis
  Columbia, 2006

---

<table>
<thead>
<tr>
<th>A</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>B</td>
</tr>
</tbody>
</table>
Why Take Programming Languages and Compilers?

To appreciate **the marriage of theory and practice**

To explore the dimensions of **computational thinking**

To exercise **creativity**

To learn **robust software development practices**
Plus Three Skills for Life

Project management

Teamwork

Communication both oral and written
Telling Lessons Learned

• “Designing a language is hard and designing a simple language is extremely hard!”

• “During this course we realized how naïve and overambitious we were, and we all gained a newfound respect for the work and good decisions that went into languages like C and Java which we’ve taken for granted for years.”