Al Aho

aho@cs.columbia.edu

Teaching Compilers

COMPUTER SCIENCE AT
COLUMBIA UNIVERSITY

SIGCSE
Milwaukee, WI
March 12, 2010

Why Take Programming Languages and Compilers?

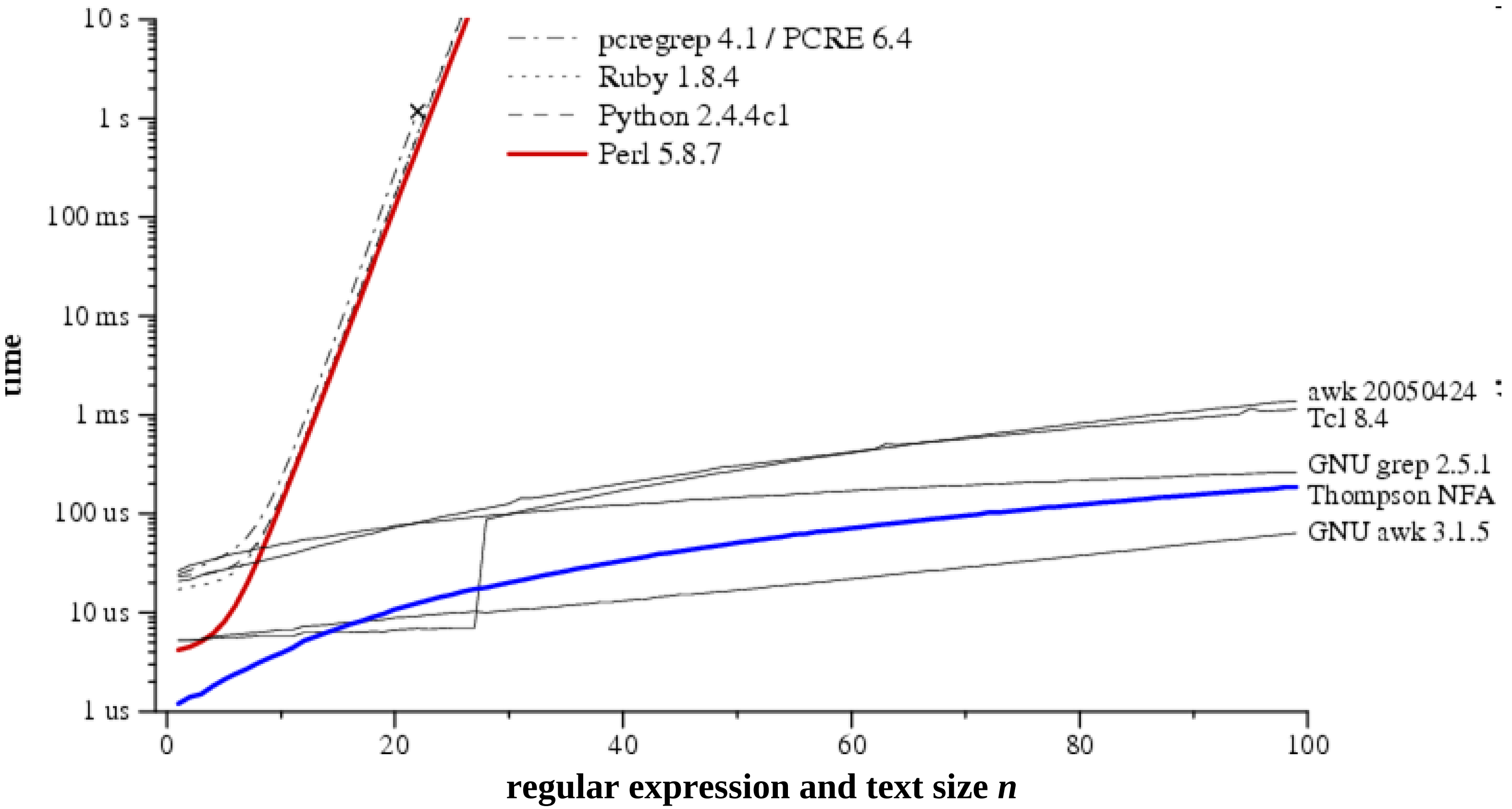
Why Take Programming Languages and Compilers?

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:

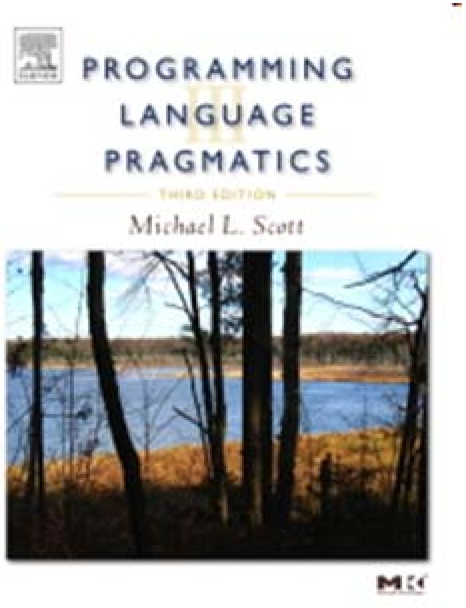
Procedural: C, C++, C#, Java

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

1970	2010
Fortran	Java
Lisp	C
Cobol	PHP
Algol 60	C++
APL	Visual Basic
Snobol 4	C#
Simula 67	Python
Basic	Perl
PL/1	Delphi
Pascal	JavaScript

[http://www.tiobe.com]

Programming Languages

Today there are thousands of programming languages.

The website 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.

.
.
.

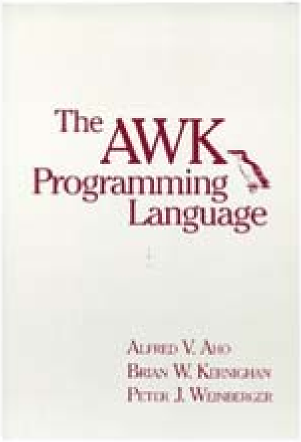
2 bottles of beer on the wall, 2 bottles of beer.
Take one down and pass it around, 1 bottle of beer on the wall.

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
}
```

[Osamu Aoki, http://people.debian.org/~osamu]

“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]

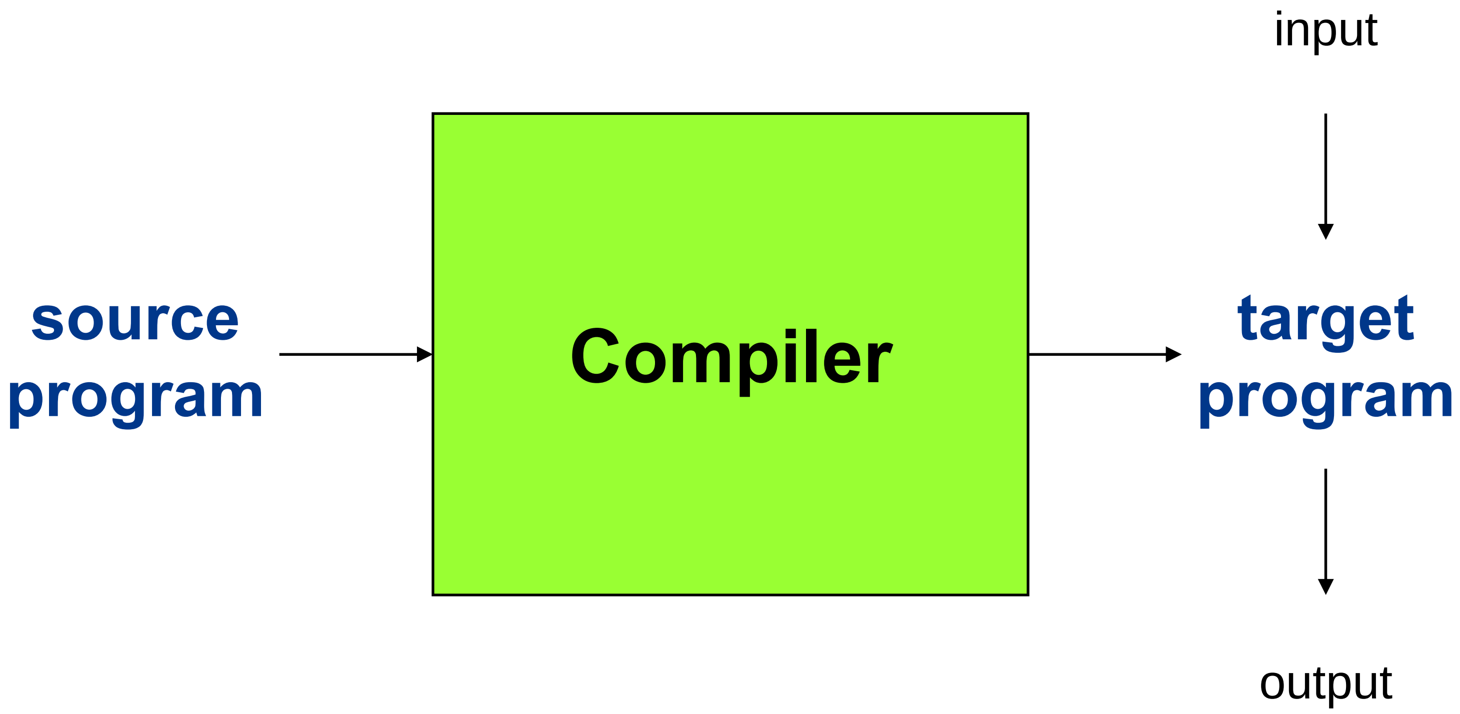
Conlangs: Made-Up Languages

Okrent lists 500 invented languages including:

- **Lingua Ignota** [Hildegaard 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?



input

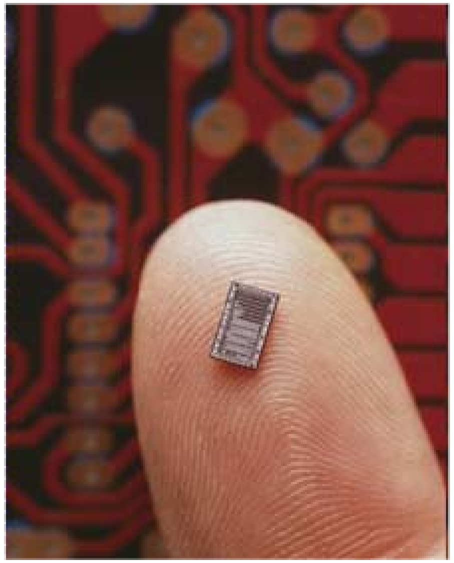
source program

Compiler

target program

output

Target Languages



Another programming language

CISCs

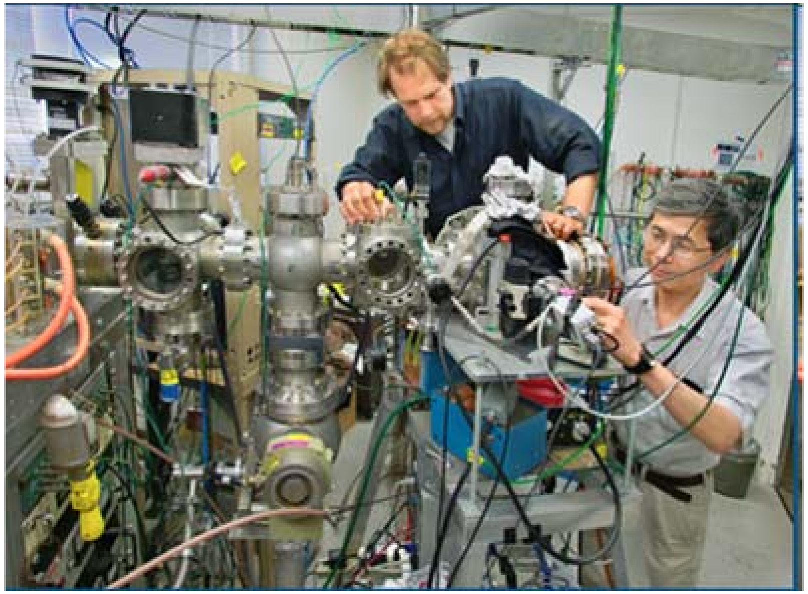
RISCs

Vector machines

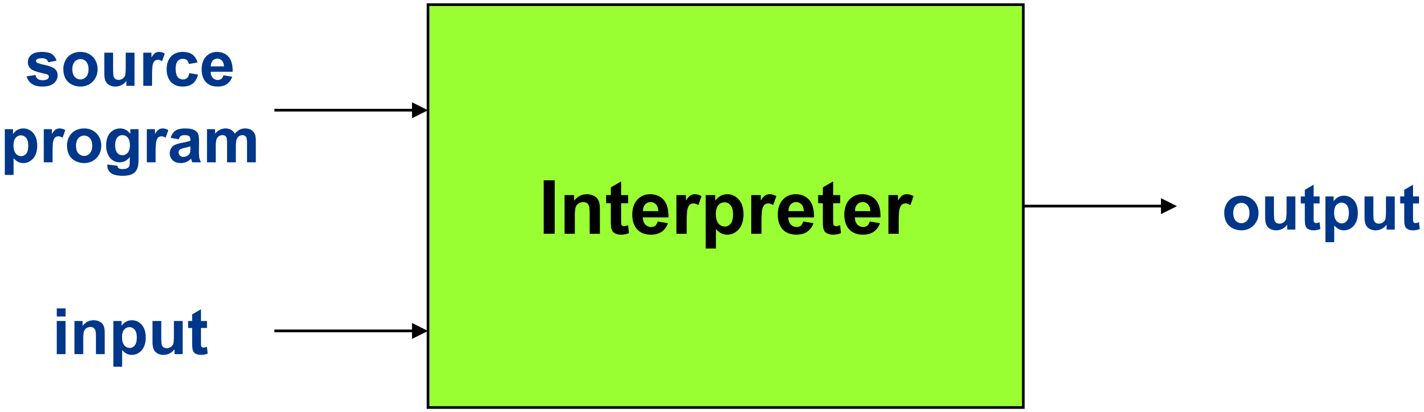
Multicores

GPUs

Quantum computers



An Interpreter Directly Executes a Source Program on its Input



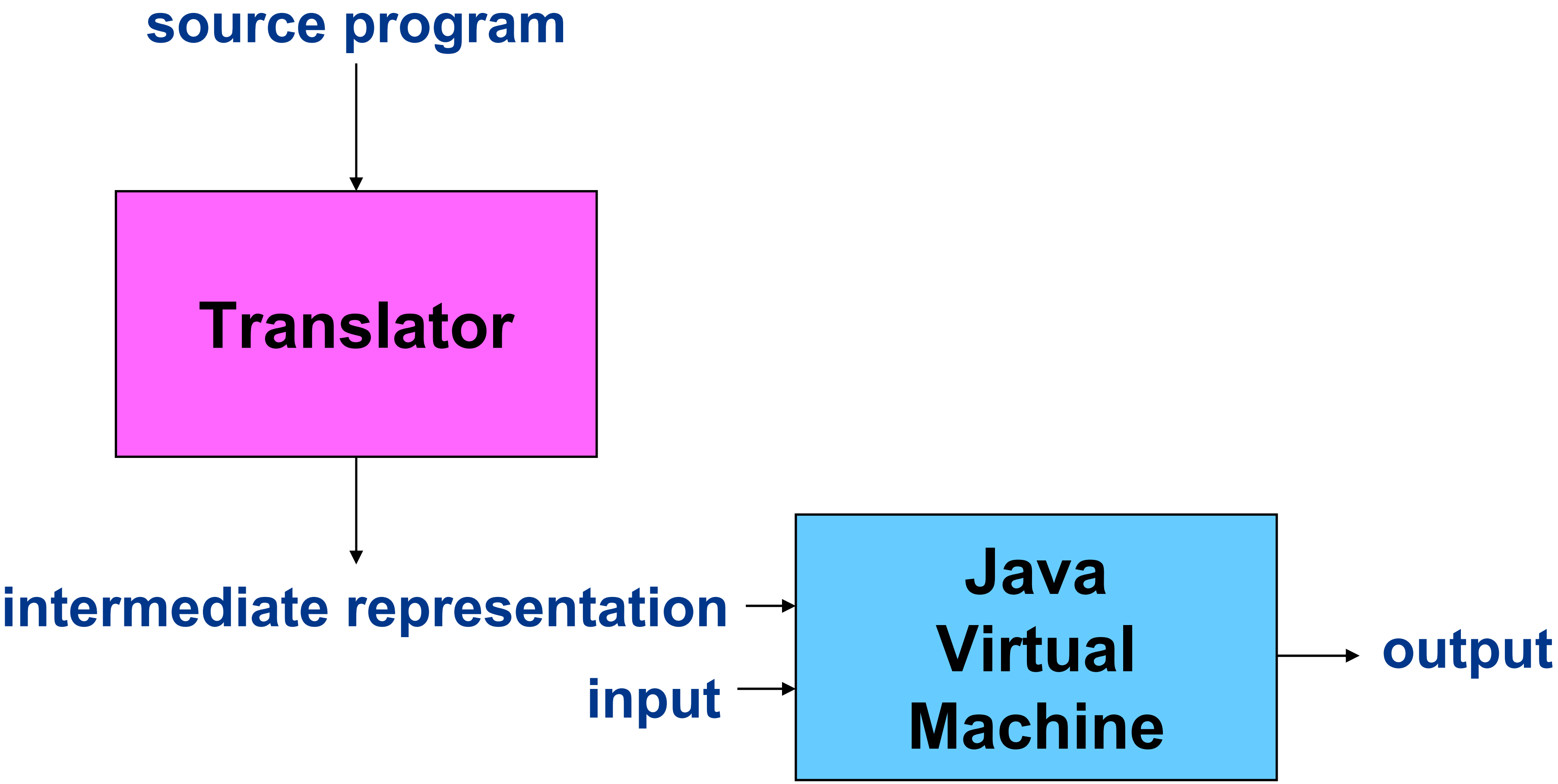
source program

input

Interpreter

output

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?

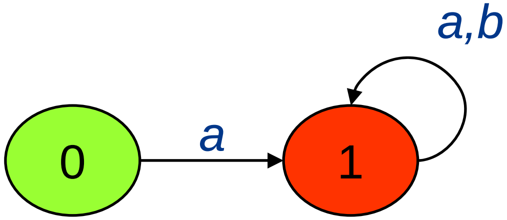
- **Unifying abstractions**
- **Fundamental models**
- **Basic algorithms**
- **Computational thinking**

Specifying Syntax: Regular Expressions and Finite Automata

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.

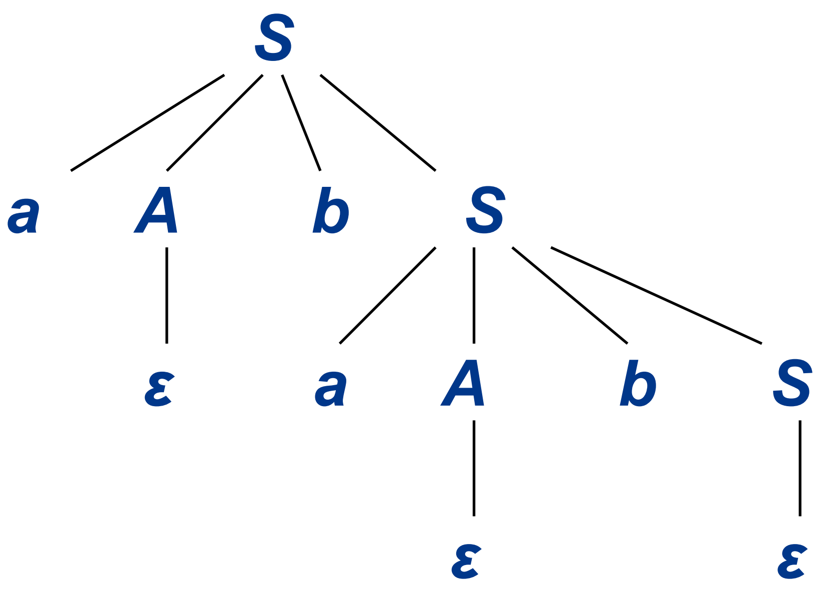
Specifying Syntax: Context-free Grammars

This grammar *G* generates all strings of *a*'s and *b*'s with the same number of *a*'s as *b*'s:

$S \rightarrow aAbS \mid bBaS \mid \varepsilon$

$A \rightarrow aAbA \mid \varepsilon$

$B \rightarrow bBaB \mid \varepsilon$

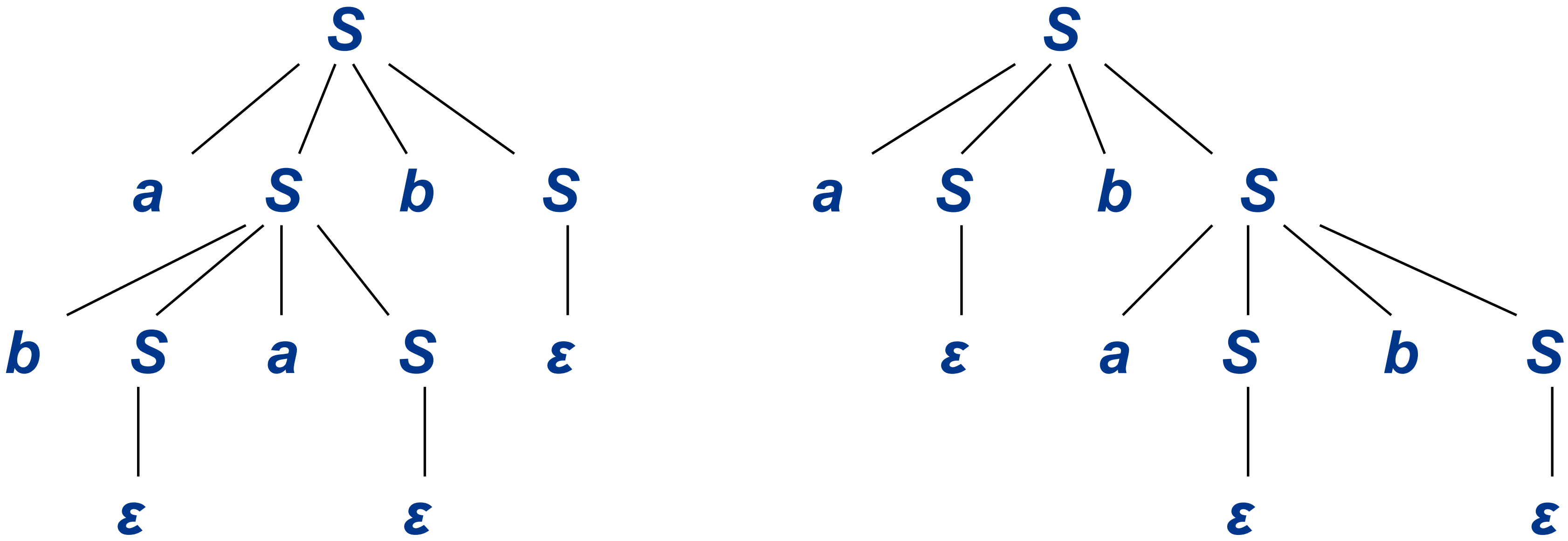


G is unambiguous and has only one parse tree for every sentence in L(G).

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 $\dfrac{1}{n+1}\dbinom{2n}{n}$ 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

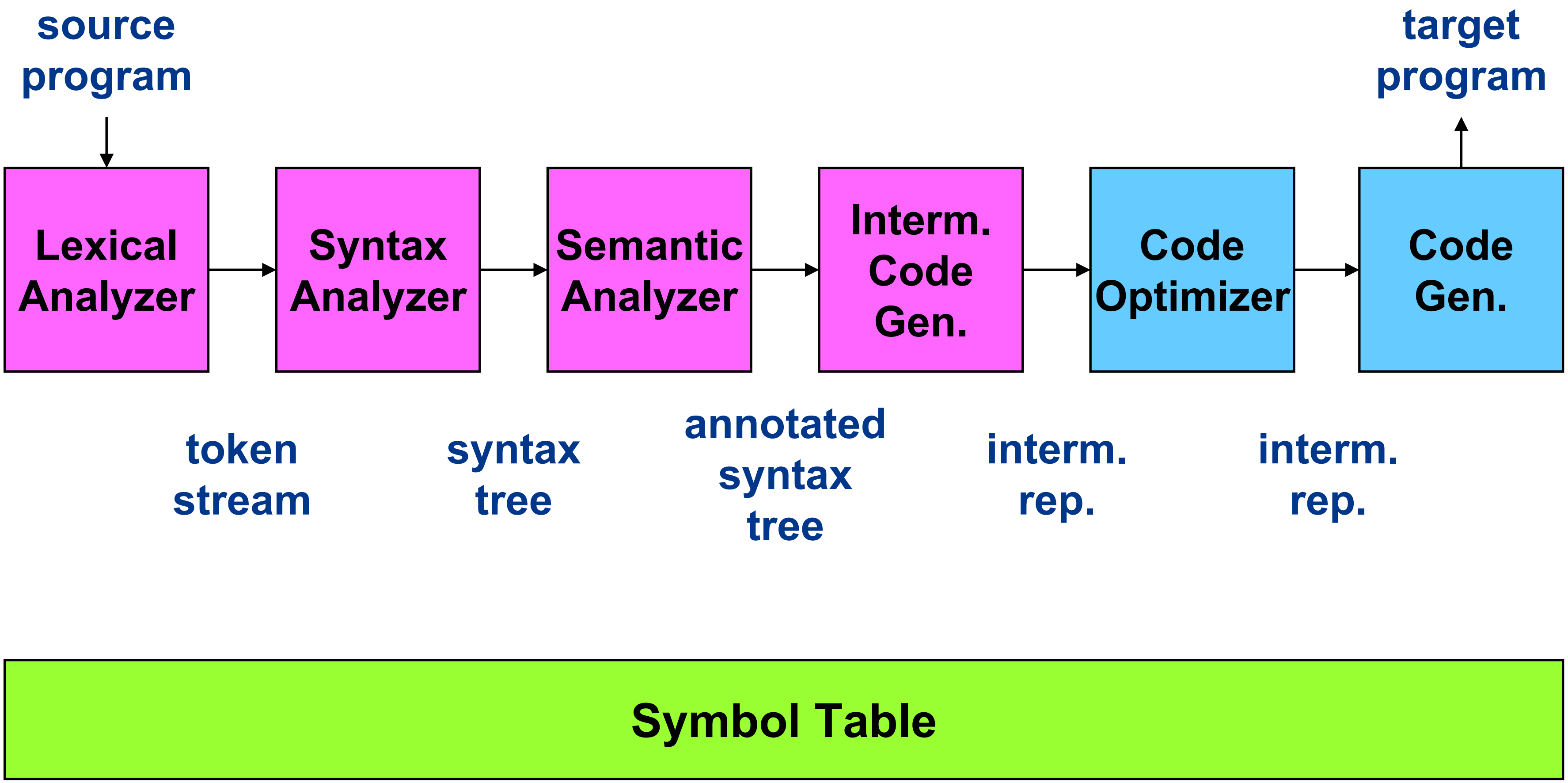
Denotational semantics

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

Lexical Analyzer

token stream

Syntax Analyzer

syntax tree

Semantic Analyzer

annotated syntax tree

Interm. Code Gen.

interm. rep.

Code Optimizer

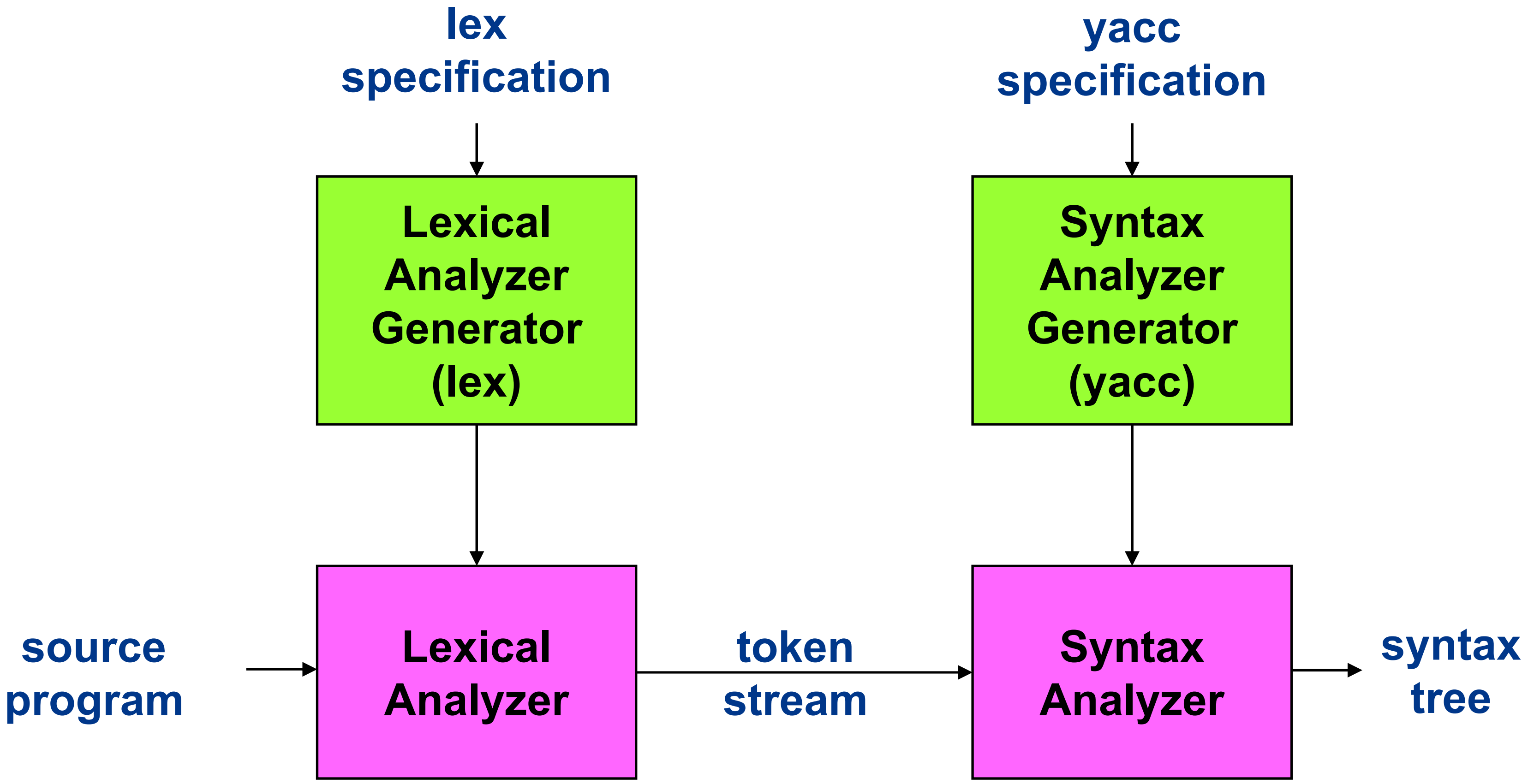
interm. rep.

Code Gen.

target program

Symbol Table

Compiler Component Generators



lex specification

Lexical Analyzer Generator (lex)

yacc specification

Syntax Analyzer Generator (yacc)

source program

Lexical Analyzer

token stream

Syntax Analyzer

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

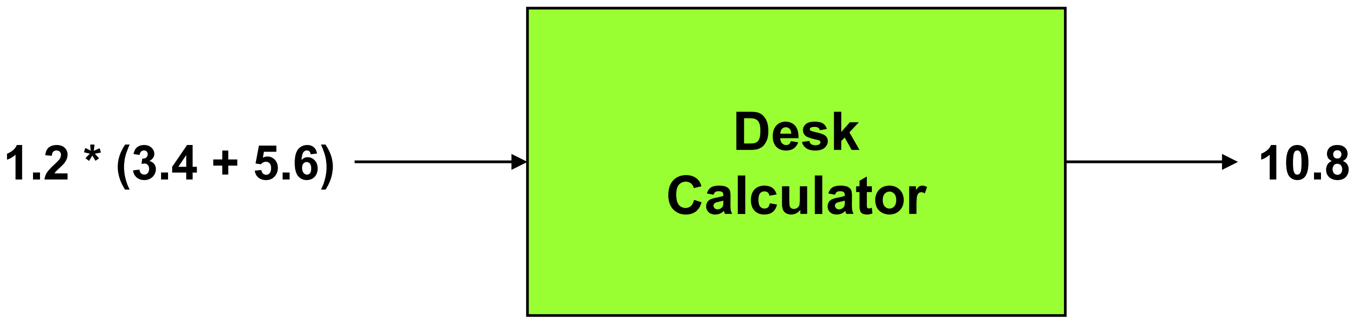
```
%token NUMBER
%left '+'
%left '*'
%%
lines : lines expr '\n' { printf("%g\n", $2); }
      | /* empty */
      ;
expr  : expr '+' expr   { $$ = $1 + $3; }
      | expr '*' expr   { $$ = $1 * $3; }
      | '(' expr ')'    { $$ = $2; }
      | NUMBER
      ;
%%
#include "lex.yy.c"
```

Creating the Desk Calculator

Invoke the commands

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

Result



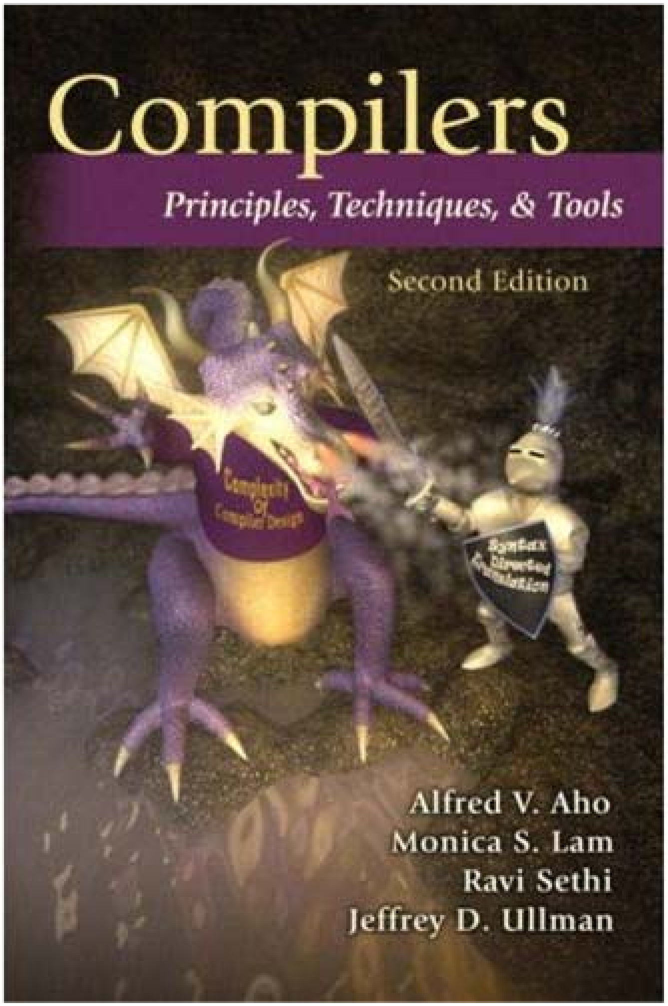
1.2 * (3.4 + 5.6)

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: a Stream Processing Language

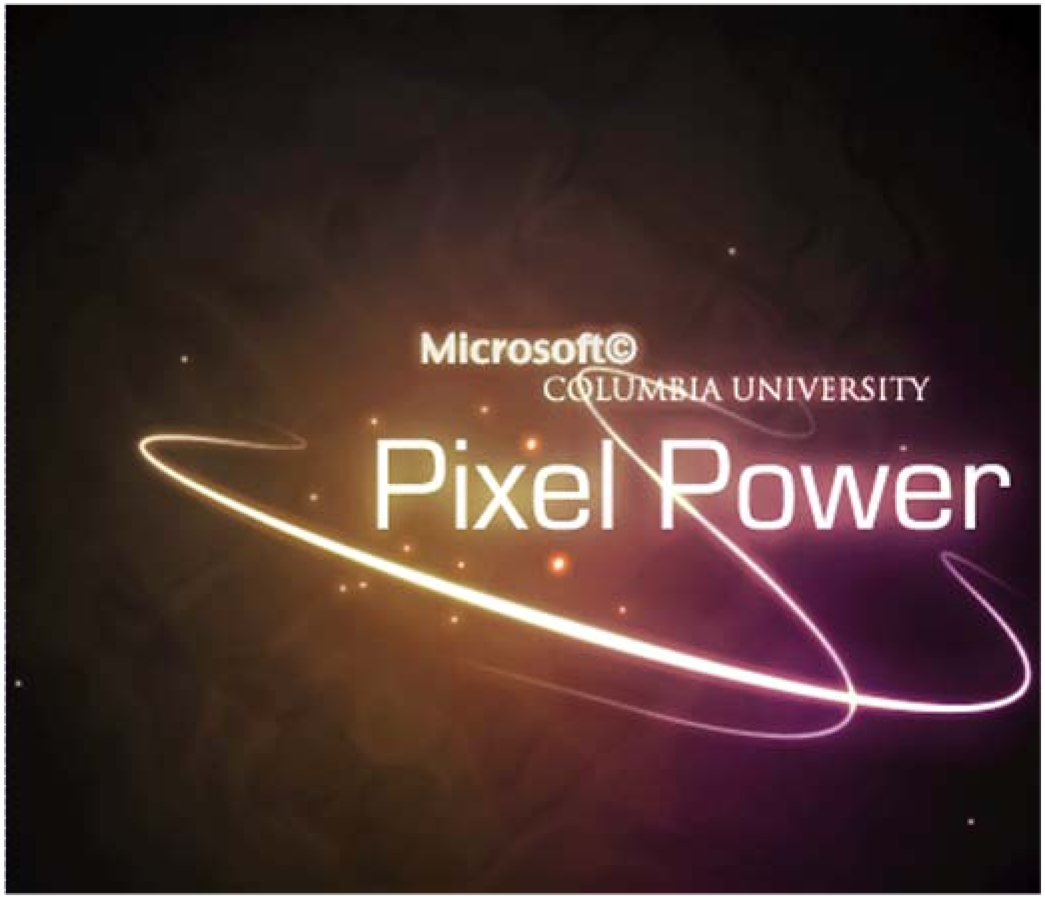
Eliane Kabkab

Nageswar Keetha

Eric Liu

Kaushik Viswanathan

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

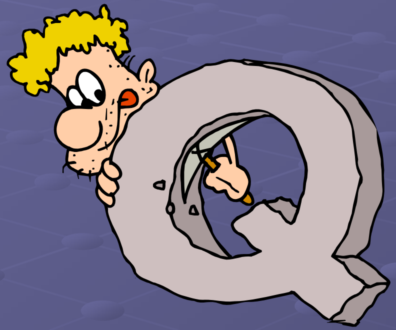
- Communication: after every class, bulletin board discussion, shared meeting notes & documents
- Loss of team member: redistribute jobs, pare down grammar
- The best ways for getting along in a new team may not get things done. Push ahead.

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



Q-HSK

A Quantum Programming Language

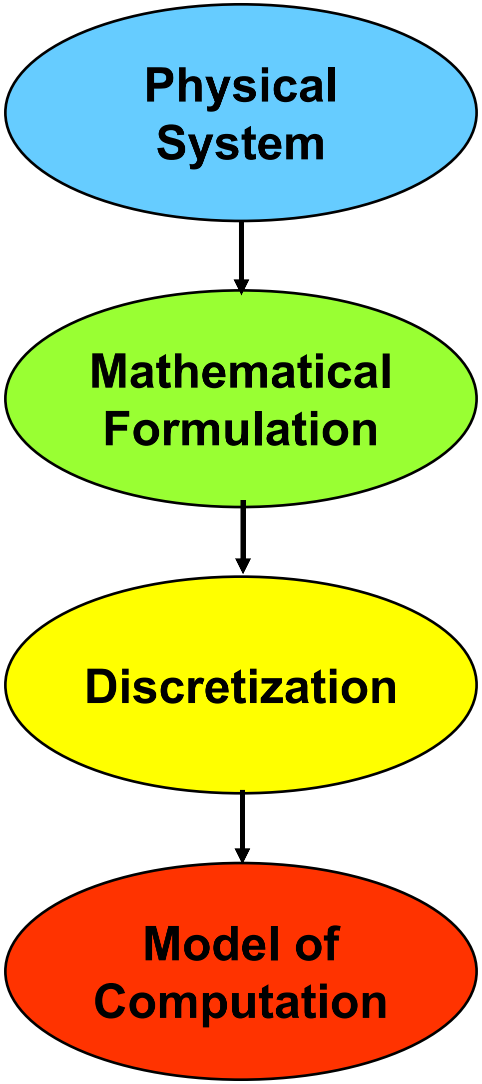
Presented by Team HSK:

Katherine **H**eller, Krysta **S**vore, Maryam **K**amvar

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

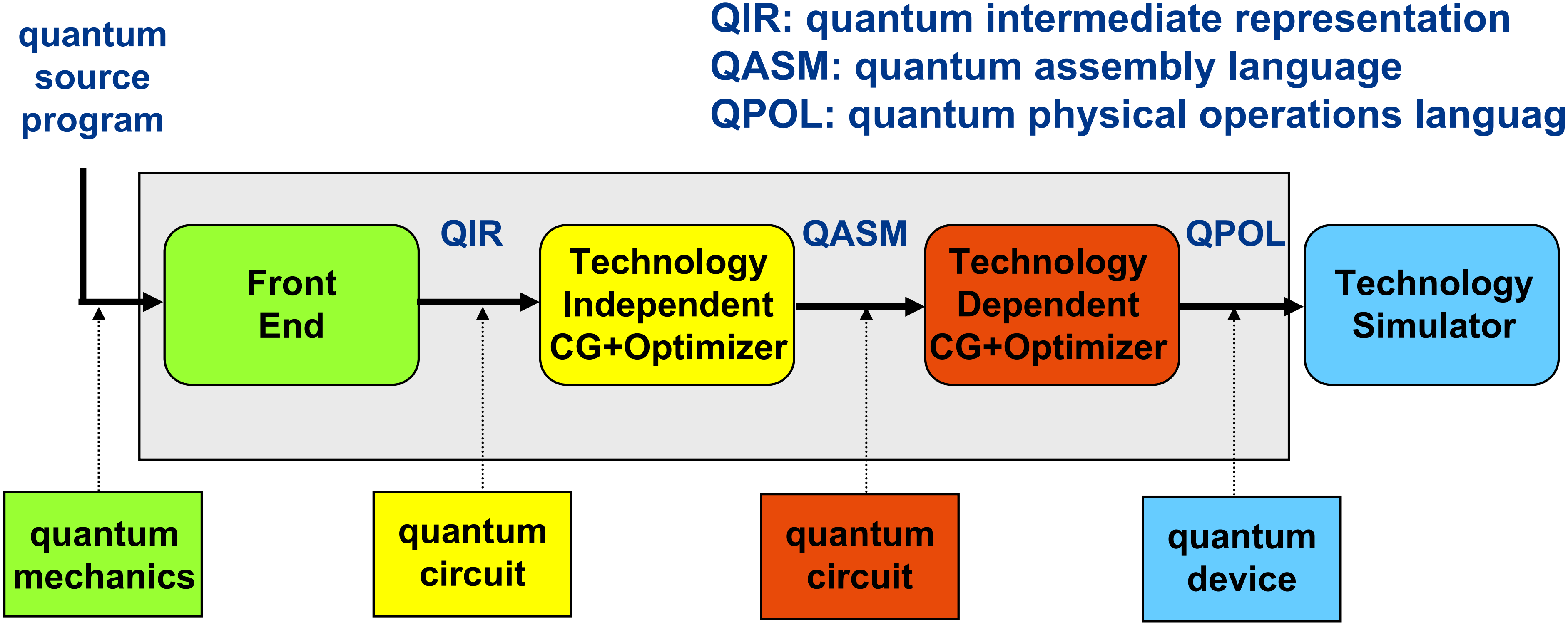
Mathematical Formulation

Discretization

Model of Computation

Quantum Computer Compiler

QIR: quantum intermediate representation
QASM: quantum assembly language
QPOL: quantum physical operations language


quantum
source
program

Front
End

QIR

Technology
Independent
CG+Optimizer

QASM

Technology
Dependent
CG+Optimizer

QPOL

Technology
Simulator

quantum
mechanics

quantum
circuit

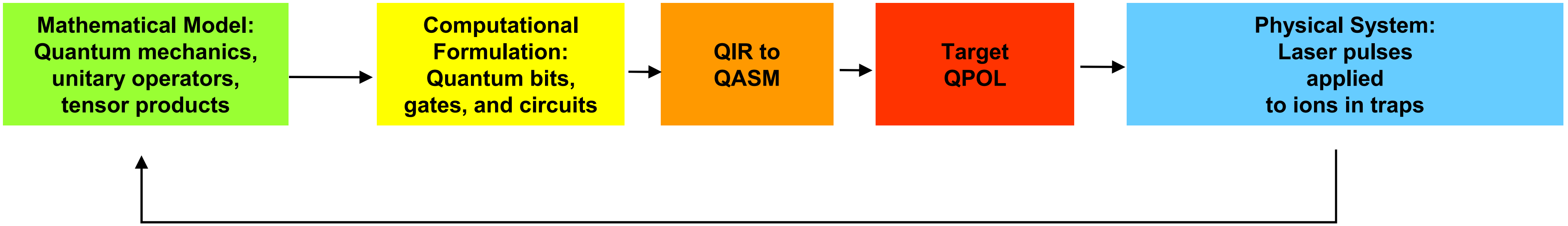
quantum
circuit

quantum
device

Computational abstractions

K. Svore, A. Aho, A. Cross, I. Chuang, I. Markov
A Layered Software Architecture for Quantum Computing Design Tools
IEEE Computer, 2006, vol. 39, no. 1, pp.74-83

QCC for Ion Trap Quantum Computing Device



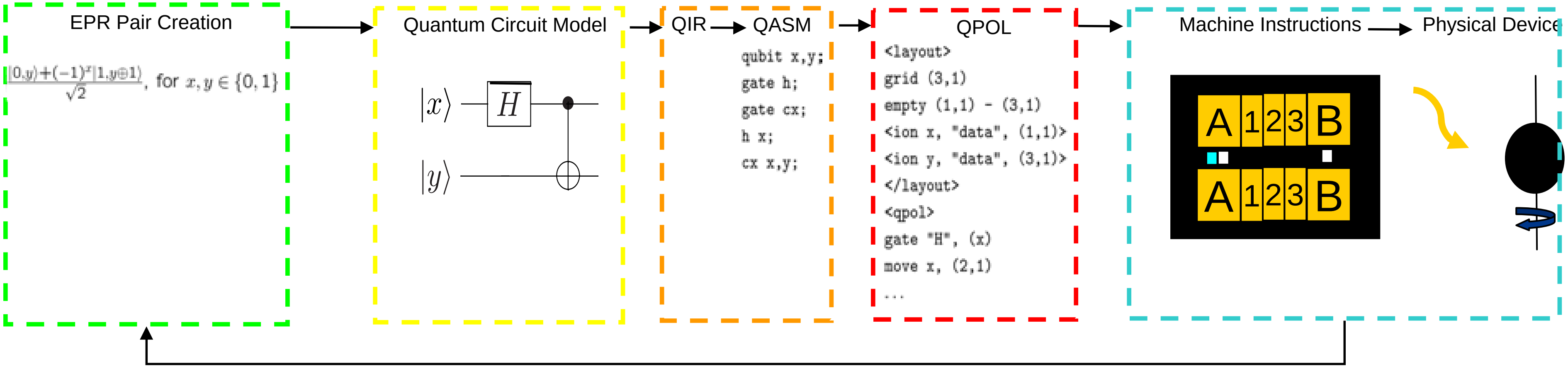
Mathematical Model: Quantum mechanics, unitary operators, tensor products

Computational Formulation: Quantum bits, gates, and circuits

QIR to QASM

Target QPOL

Physical System: Laser pulses applied to ions in traps



EPR Pair Creation

$\frac{|0,y\rangle+(-1)^x|1,y\oplus1\rangle}{\sqrt{2}}$, for $x, y \in \{0,1\}$

Quantum Circuit Model

QIR → QASM

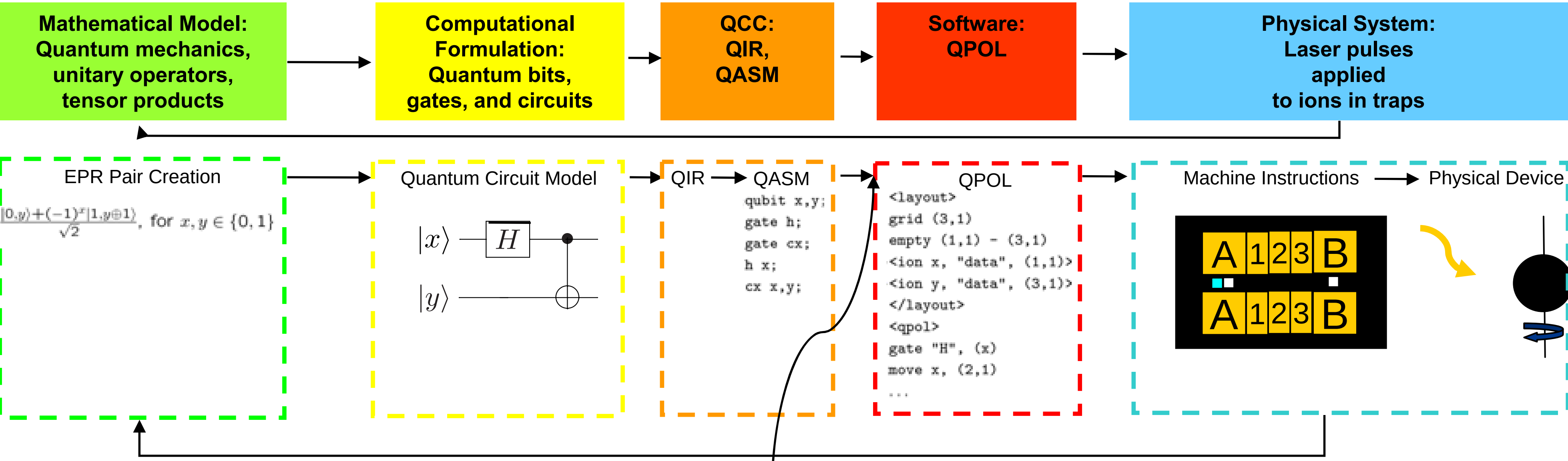
```
qubit x,y;
gate h;
gate cx;
h x;
cx x,y;
```

QPOL

```
<layout>
grid (3,1)
empty (1,1) - (3,1)
<ion x, "data", (1,1)>
<ion y, "data", (3,1)>
</layout>
<qpol>
gate "H", (x)
move x, (2,1)
...
```

Machine Instructions → Physical Device

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

EPR Pair Creation

$\frac{|0,y\rangle + (-1)^x|1,y\oplus 1\rangle}{\sqrt{2}}$, for $x, y \in \{0,1\}$

Quantum Circuit Model

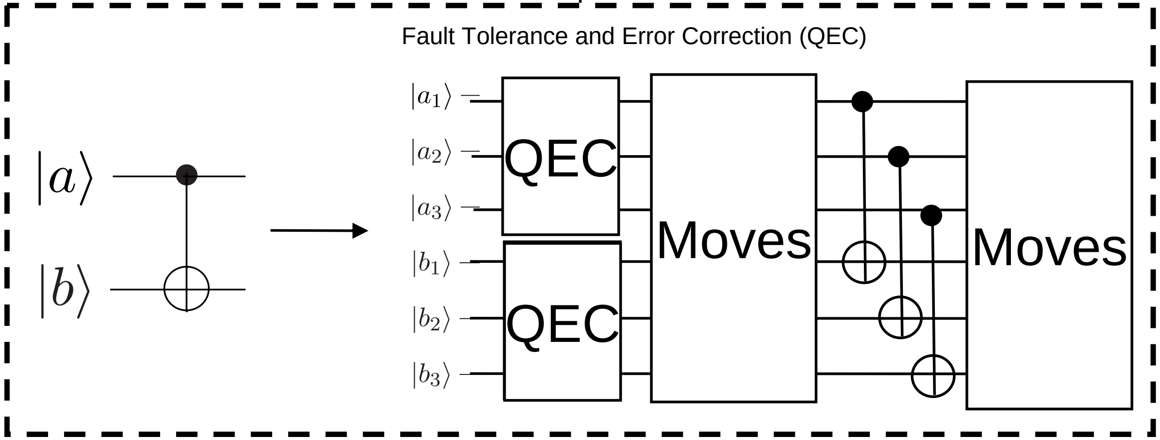
QIR → QASM

```
qubit x,y;
gate h;
gate cx;
h x;
cx x,y;
```

QPOL

```
<layout>
grid (3,1)
empty (1,1) - (3,1)
<ion x, "data", (1,1)>
<ion y, "data", (3,1)>
</layout>
<qpol>
gate "H", (x)
move x, (2,1)
...
```

Machine Instructions → Physical Device



Fault Tolerance and Error Correction (QEC)

K. Svore
PhD Thesis
Columbia, 2006

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.”**