Inspired by Stanford John Mitchell CS’242
Prerequisites

• Software Project
• Computational models
Textbooks

- J. Mitchell. Concepts in Programming Languages
- B. Pierce. Types and Programming Languages
- B. Tate. Seven Languages in Seven Weeks
Course Grade

• 50% Assignments (8 assignments)
  – 2 person teams

• 50% Exam
  – Must pass exam
  – Graduate students can take home exam
    • More difficult
Goals

• Learn about cool programming languages
• Learn about useful programming languages
• Understand theoretical concepts in programming languages
• Become a better programmer in your own programming language
• Have fun
Course Goals (Cont)

• Programming Language Concepts
  – A language is a “conceptual universe” (Perlis)
    • Framework for problem-solving
    • Useful concepts and programming methods
  – Understand the languages you use, by comparison
  – Appreciate history, diversity of ideas in programming
  – Be prepared for new programming methods, paradigms, tools

• Critical thought
  – Identify properties of language, not syntax or sales pitch

• Language and implementation
  – Every convenience has its cost
    • Recognize the cost of presenting an abstract view of machine
    • Understand trade-offs in programming language design
Language goals and trade-offs

Architect

Compiler, Runtime environment

Programmer

Testing

Diagnostic Tools

Programming Language
What’s new in programming languages

• Commercial trend over past 5+ years
  – Increasing use of type-safe languages: Java, C#, Scala
  – Scripting languages, other languages for web applications

• Teaching trends
  – Java replaced C as most common intro language
    • Less emphasis on how data, control represented in machine

• Research and development trends
  – Modularity
    • Java, C++: standardization of new module features
  – Program analysis
    • Automated error detection, programming env, compilation
  – Isolation and security
    • Sandboxing, language-based security, ...
  – Web 2.0
    • Increasing client-side functionality, mashup isolation problems
What’s worth studying?

- Dominant languages and paradigms
  - Leading languages for general systems programming
  - Explosion of programming technologies for the web
- Important implementation ideas
- Performance challenges
  - Concurrency
- Design tradeoffs
- Concepts that research community is exploring for new programming languages and tools
- Formal methods in practice
  - Grammars
  - Semantics
  - Types and Type Systems
...
Related Courses

• Seminar in programming Language
• Compilers
• Semantics of programming languages
• Program analysis
• Software Verification
The Fortran Programming Language

- FORMula TRANslating System
- Designed in early 50s by John Backus from IBM
  - Turing Award 1977
  - Responsible for Backus Naur Form (BNF)
- Intended for Mathematicians/Scientists
- Still in use
Lisp

• The second-oldest high-level programming language
• List Processing Language
• Designed by John McCarty 1958
  – Turing Award for Contributions to AI
• Influenced by Lambda Calculus
• Pioneered the ideas of tree data structures, automatic storage management, dynamic typing, conditionals, higher-order functions, recursion, and the self-hosting compiler
Lisp Design Flaw: Dynamic Scoping

```lisp
procedure p;
    var x: integer
    procedure q;
        begin { q }
        ...
        x
        ...
        end { q };
    procedure r;
    var x: integer
    begin { r }
    q;
    q;
    end; { r }
begin { p }
    q;
    r;
end { p }
```
The Algol 60

- ALGOrithmic Language 1960
- Designed by Researchers from Europe/US
- Led by Peter Naur 2005 Turing Award
- Pioneered: Scopes, Procedures, Static Typing

<table>
<thead>
<tr>
<th>Name</th>
<th>Year</th>
<th>Author</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1 ALGOL 60</td>
<td>1960</td>
<td>Dijkstra and Zonneveld</td>
<td>Netherlands</td>
</tr>
<tr>
<td>Algol</td>
<td>1960</td>
<td>Irons</td>
<td>USA</td>
</tr>
<tr>
<td>Burroughs Algol</td>
<td>1961</td>
<td>Burroughs</td>
<td>USA</td>
</tr>
<tr>
<td>Case ALGOL</td>
<td>1961</td>
<td></td>
<td>USA</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Algol Design Flaw: Power

• $E ::= \text{ID} \mid \text{NUM} \mid E + E \mid E - E \mid E \times E \mid E \div E \mid E \^{} E$


C Programming Language

- Statically typed, general purpose systems programming language
- Computational model reflects underlying machine
- Designed by Dennis Ritchie, ACM Turing Award for Unix
- (Initial) Simple and efficient one pass compiler
- Replaces assembly programming
- Widely available
- Became widespread
Simple C design Flaw

- Switch cases without breaks continue to the next case

```c
switch (e) {
    case 1: x = 1;
    case 2: x = 4;
        break;
    default: x = 8;
}
```
A Pathological C Program

```c
a = malloc(...) ;
b = a;
free (a);
c = malloc (...);
if (b == c) printf(“unexpected equality”);
```
Conflicting Arrays with Pointers

• An array is treated as a pointer to first element
• E1[E2] is equivalent to ptr dereference: *(*(E1)+(E2))
• a[i] == i[a]
• Programmers can break the abstraction
• The language is not type safe
  – Even stack is exposed
Buffer Overrun Exploits

```c
void foo (char *x) {
    char buf[2];
    strcpy(buf, x);
}

int main (int argc, char *argv[]) {
    foo(argv[1]);
}
```

> ./a.out abracadabra
Segmentation fault
Buffer Overrun Exploits

```c
int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[16];

    strcpy(password_buffer, password);
    if(strcmp(password_buffer, "brillig") == 0) auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0) auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
    if(check_authentication(argv[1])) {
        printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf(" Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
    } else
        printf("\nAccess Denied.\n");
}
```

(source: “hacking – the art of exploitation, 2nd Ed”)
Exploiting Buffer Overruns

evil input → Application → Something really bad happens
Summary C

• Unsafe
• Exposes the stack frame
  – Parameters are computed in reverse order
• Hard to generate efficient code
  – The compiler need to prove that the generated code is correct
  – Hard to utilize resources
• Ritchie quote
  – “C is quirky, flawed, and a tremendous success”
ML programming language

- Statically typed, general-purpose programming language
  - “Meta-Language” of the LCF theorem proving system
- Designed in 1973
- Type safe, with formal semantics
- Compiled language, but intended for interactive use
- Combination of Lisp and Algol-like features
  - Expression-oriented
  - Higher-order functions
  - Garbage collection
  - Abstract data types
  - Module system
  - Exceptions
  - Encapsulated side-effects

Robin Milner, ACM Turing-Award for ML, LCF Theorem Prover, ...
Haskell

• Haskell programming language is
  – Similar to ML: general-purpose, strongly typed, higher-order, functional, supports type inference, interactive and compiled use
  – Different from ML: lazy evaluation, purely functional core, rapidly evolving type system

• Designed by committee in 80’s and 90’s to unify research efforts in lazy languages
  – Haskell 1.0 in 1990, Haskell ‘98, Haskell’ ongoing
  – “A History of Haskell: Being Lazy with Class” HOPL 3
Language Evolution

Many others: Algol 58, Algol W, Scheme, EL1, Mesa (PARC), Modula-2, Oberon, Modula-3, Fortran, Ada, Perl, Python, Ruby, C#, Javascript, F#, Scala...
The quick death

Most Research Languages

Practitioners

Geeks

1yr 5yr 10yr 15yr
Successful Research Languages

The slow death
Geeks

Practitioners

C++, Java, Perl, Ruby

Threshold of immortality

The complete absence of death

1,000,000

10,000

100

1

1yr 5yr 10yr 15yr
“Learning Haskell is a great way of training yourself to think functionally so you are ready to take full advantage of C# 3.0 when it comes out” (blog Apr 2007)

“I'm already looking at coding problems and my mental perspective is now shifting back and forth between purely OO and more FP styled solutions” (blog Mar 2007)
Programming Language Paradigms

• Imperative
  – Algol, PL1, Fortran, Pascal, Ada, Modula, and C
  – Closely related to “von Neumann” Computers

• Object-oriented
  – Simula, Smalltalk, Modula3, C++, Java, C#, Python
  – Data abstraction and ‘evolutionary’ form of program development
    • Class An implementation of an abstract data type (data+code)
    • Objects Instances of a class
    • Fields Data (structure fields)
    • Methods Code (procedures/functions with overloading)
    • Inheritance Refining the functionality of a class with different fields and methods

• Functional
  – Lisp, Scheme, ML, Miranda, Hope, Haskel, OCaml, F#

• Functional/Imperative
  – Rubby

• Logic Programming
  – Prolog
Other Languages

• Hardware description languages
  – VHDL
  – The program describes Hardware components
  – The compiler generates hardware layouts

• Scripting languages
  – Shell, C-shell, REXX, Perl
  – Include primitives constructs from the current software environment

• Web/Internet
  – HTML, Telescript, JAVA, Javascript

• Graphics and Text processing
  TeX, LaTeX, postscript
  – The compiler generates page layouts

• Domain Specific
  – SQL
  – yacc/lex/bison/awk

• Intermediate-languages
  – P-Code, Java bytecode, IDL, CLR
What make PL successful?

- Beautiful syntax
- Good design
- Good productivity
- Good performance
- Safety
- Portability
- Good environment
  - Compiler
  - Interpreter
- Influential designers
- Solves a need
  - C efficient system programming
  - Javascript Browsers
Instructor’s Background

• First programming language Pascal
• Soon switched to C (unix)
  • Efficient low level programming was the key
  • Small programs did amazing things
• Led a big project was written in common lisp
  • Semi-automatically port low level IBM OS code between 16 and 32 bit architectures
• The programming setting has dramatically changed:
  • Object oriented
  • Garbage collection
  • Huge programs
  • Performance depends on many issues
  • Productivity is sometimes more importance than performance
  • Software reuse is a key
Other Lessons Learned

• Futuristic ideas may be useful problem-solving methods now, and may be part of languages you use in the future
  • Examples
    • Recursion
    • Object orientation
    • Garbage collection
    • High level concurrency support
    • Higher order functions
    • Pattern matching
More examples of practical use of futuristic ideas

- Function passing: pass functions in C by building your own closures, as in STL “function objects”
- Blocks are a nonstandard extension added by Apple to C that uses a lambda expression like syntax to create closures
- Continuations: used in web languages for workflow processing
- Monads: programming technique from functional programming
- Concurrency: atomicity instead of locking
- Decorators in Python to dynamically change the behavior of a function
- Mapreduce for distributed programming
Unique Aspects of PL

• The ability to formally define the syntax of a programming language
• The ability to formally define the semantics of the programming language (operational, axiomatic, denotational)
• The ability to prove that a compiler/interpreter is correct
• Useful concepts: Closures, Monads, Continuations, ...
Theoretical Topics Covered

• Syntax of PLs
• Semantics of PLs
  – Operational Semantics
  – $\lambda$ calculus
• Program Verification
  – Floyd-Hoare style verification
• Types
Languages Covered

• Python (Used but not taught)
• ML (Ocaml)
• Javascript
• Scala
• Go & Cloud computing
Interesting Topics not covered

• Concurrency
• Modularity
• Object orientation
• Aspect oriented
• Garbage collection
• Virtual Machines
• Compilation techniques
# Part 1: Principles

<table>
<thead>
<tr>
<th>Date</th>
<th>Lecture</th>
<th>Targil</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>7/3</td>
<td>Syntax of Programming Languages</td>
<td>Recursive Decent</td>
<td>Ex. 1 – Syntax</td>
</tr>
<tr>
<td>14/3</td>
<td>Natural Operational Semantics</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>21/3</td>
<td>Small Step Operational Semantics (SOS)</td>
<td>=</td>
<td>Ex. 2 – Semantics</td>
</tr>
<tr>
<td>28/3</td>
<td>A Crash Course in Logic and induction</td>
<td>=</td>
<td>Ex. 3 – Logic</td>
</tr>
<tr>
<td>4/4</td>
<td>Floyd-Hoare Verification</td>
<td>Dafny</td>
<td>Ex 4 -- Dafny</td>
</tr>
<tr>
<td>11/4</td>
<td>Lambda Calculus</td>
<td>Lambda Calculus</td>
<td>Ex 5 – Lambda Calculus</td>
</tr>
</tbody>
</table>
# Part 2: Applications

<table>
<thead>
<tr>
<th>Date</th>
<th>Lecture</th>
<th>Targil</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>2/5</td>
<td>Basic ML</td>
<td>Basic ML</td>
<td>Ex 6 – ML Project</td>
</tr>
<tr>
<td>9/5</td>
<td>Advanced ML</td>
<td>Advanced ML</td>
<td></td>
</tr>
<tr>
<td>16/5</td>
<td>Type Inference</td>
<td>Type Inference</td>
<td></td>
</tr>
<tr>
<td>23/5</td>
<td>Basic Javascript</td>
<td>Basic Javascript</td>
<td>Ex. 7 – JavaScript Project</td>
</tr>
<tr>
<td>30/5</td>
<td>Advanced Javascript</td>
<td>Jquery</td>
<td></td>
</tr>
<tr>
<td>6/6</td>
<td>Scala</td>
<td>Scala</td>
<td>Ex. 8 – Scala Project</td>
</tr>
</tbody>
</table>
Summary

• Learn cool programming languages
• Learn useful programming language concepts
• But be prepared to program
  – Public domain software