Programming Language Concepts

Mooly Sagiv
msagiv@acm.org
Tuesday 11-13, Schriber 317
TA: Oded Padon
Email: odedp@mail.tau.ac.il

http://www.cs.tau.ac.il/~msagiv/courses/pl18.html

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
• Semantics with Applications by Flemming Nielson and Hanne Riis Nielson
• Real World Ocaml by Anil Madhavapeddy, Jason Hickey, and Yaron Minsky
• JavaScript: The Good Parts by Douglas Crockford
Course Grade

• 30% Assignments (5 assignments)
  – 2-3 person teams

• 70% Exam
  – Must pass exam
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

Programmer

Testing

Compiler, Runtime environment

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: JavaScript

• 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
# New PL in Industry

1. Performance
2. Security
3. Productivity

<table>
<thead>
<tr>
<th>PL</th>
<th>Company</th>
<th>Predecessor</th>
<th>Domain</th>
<th>Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go</td>
<td>Google</td>
<td>C</td>
<td>Systems</td>
<td>Channels Type Inference ML modules GC</td>
</tr>
<tr>
<td>Dart</td>
<td>Google</td>
<td>Javascript</td>
<td>Web</td>
<td>Types</td>
</tr>
<tr>
<td>Rust</td>
<td>Firefox</td>
<td>C++</td>
<td>System Browser</td>
<td>Type Inference Ownership Linear Types</td>
</tr>
<tr>
<td>Hack</td>
<td>Facebook</td>
<td>PHP</td>
<td>Social Network</td>
<td>Gradual Types</td>
</tr>
</tbody>
</table>
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

• Compilers
• Semantics of programming languages
• Program analysis
• Model checking
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

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

- $E ::= \text{ID} \mid \text{NUM} \mid E + E \mid E - E \mid E \times E \mid E / E \mid E^{**} E$
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;
}
```
Type Safety

• A programming language is type safe if every well typed program has no undefined semantics

• No runtime surprise

• Is C type safe?

• How about Java?
void foo(s) {
    char c[100];
    strcpy(c, s);
}

Breaking Safety in C
A Pathological C Program

```c
a = malloc(...) ;
b = a;
free (a);
c = malloc (...);
if (b == c) printf(“unexpected equality”);
```
Another Pathological C Program

```c
union {
    int x;
    int *p;
} mixed;
mixed.x = 1700;
free(mixed.p);
```
Conflicting Arrays with Pointers

• An array is treated as a pointer to first element (syntactic sugar)

• E1[E2] is equivalent to ptr dereference: 
  \[ E1[E2] == *((E1)+(E2)) \]

• 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]);
}
```

source code

```
> ./a.out abracadabra
Segmentation fault
```

terminal
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");
}
Exploiting Buffer Overruns

evil input

Application

AAAAAA

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”
The Java Programming Language

- Designed by Sun 1991-95
- Statically typed and type safe
- Clean and Powerful libraries
- Clean references and arrays
- Object Oriented with single inheritance
- Interfaces with multiple inheritance
- **Portable with JVM**
- Effective JIT compilers
- Support for concurrency
- Useful for Internet
Java Critique

• Downcasting reduces the effectiveness of static type checking
  – Many of the interesting errors caught at runtime
    • Still better than C, C++

• Huge code blowouts
  – Hard to define domain specific knowledge
  – A lot of boilerplate code
  – Sometimes OO stands in our way
  – Generics only partially helps
  – Array subtype does not work
Scala

- Scala is an object-oriented and functional language which is completely interoperable with Java (.NET)
- Remove some of the more arcane constructs of these environments and adds instead:
  1. a uniform object model,
  2. pattern matching and higher-order functions,
  3. novel ways to abstract and compose programs
The Kotlin Programming Language

- Combines OO and Functional Programming
- Interoperated with JVM
- Higher order programming
- Type inference
- Memory safety
- Officially supported in Android
The RUST programming language

• Motivated by Browser development
• Type safety & Efficiency with explicit memory managements
• Built on ownership types
  – Controls aliasing
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
Many others: Algol 58, Algol W, Scheme, EL1, Mesa (PARC), Modula-2, Oberon, Modula-3, Fortran, Ada, Perl, Python, Ruby, C#, Javascript, F#, Scala, go
Most Research Languages

Practitioners

Geeks

The quick death

1yr 5yr 10yr 15yr
Successful Research Languages

Graph showing the number of practitioners and geeks over time. The graph indicates a slow death of practitioners over the years.
C++, Java, Perl, Ruby

The complete absence of death

Threshold of immortality
“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)
JavaScript History

• Developed by Brendan Eich at Netscape, 1995
  – Scripting language for Navigator 2
• Later standardized for browser compatibility
  – ECMAScript Edition 3 (aka JavaScript 1.5) -> ES5, ...
• Related to Java in name only
  – Name was part of a marketing deal
• Various implementations available
  – Spidermonkey interactive shell interface
  – Rhino: http://www.mozilla.org/rhino/
  – Browser JavaScript consoles
Motivation for JavaScript

• Netscape, 1995
  – Netscape > 90% browser market share
  – Opportunity to do “HTML scripting language”
  – Brendan Eich
    “I hacked the JS prototype in ~1 week in May
    And it showed!  Mistakes were frozen early
    Rest of year spent embedding in browser” - ICFP talk, 2005

• Common uses of JavaScript have included:
  – Form validation
  – Page embellishments and special effects
  – Dynamic content manipulation
  – Web 2.0: functionality implemented on web client
    • Significant JavaScript applications: Gmail client, Google maps
Design goals

- Brendan Eich’s 2005 ICFP talk
  - Make it easy to copy/paste snippets of code
  - Tolerate “minor” errors (missing semicolons)
  - Simplified onclick, onmousedown, etc., event handling, inspired by HyperCard
  - Pick a few hard-working, powerful primitives
    - First class functions for procedural abstraction
    - Objects everywhere, prototype-based
  - Leave all else out!
JavaScript design

• Functions based on Lisp/Scheme
  – first-class inline higher-order functions
    function (x) { return x+1; } ≡ \( \lambda x. x+1 \)

• Objects based on Smalltalk/Self
  – var pt = {x : 10, move : function(dx){this.x += dx}}
  – Functions are also objects

• Lots of secondary issues ...
  – “In JavaScript, there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders.”
    Douglas Crockford
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>5/3</td>
<td>Overview</td>
<td>No Targil</td>
<td></td>
</tr>
<tr>
<td>12/3</td>
<td>Principles of Induction</td>
<td>Principles of Induction</td>
<td>Ex. 1 – Induction</td>
</tr>
<tr>
<td>19/3</td>
<td>Natural Operational Semantics</td>
<td>=2</td>
<td></td>
</tr>
<tr>
<td>9/4</td>
<td>Small Step Operational Semantics (SOS)</td>
<td>=</td>
<td>Ex. 2 – Semantics</td>
</tr>
<tr>
<td>16/4</td>
<td>Lambda Calculus</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>23/4</td>
<td>Typed Lambda Calculus</td>
<td>=</td>
<td>Ex3— Lambda Calculus</td>
</tr>
<tr>
<td>30/4</td>
<td></td>
<td>More lambda calculus</td>
<td></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>7/5</td>
<td>Basic ML</td>
<td>More lambda calculus</td>
<td></td>
</tr>
<tr>
<td>14/5</td>
<td>Advanced ML</td>
<td>ML</td>
<td>Ex 4— ML Project</td>
</tr>
<tr>
<td>21/5</td>
<td>Type Inference</td>
<td>ML</td>
<td></td>
</tr>
<tr>
<td>28/5</td>
<td>Basic Javascript</td>
<td>Type Inference</td>
<td></td>
</tr>
<tr>
<td>4/6</td>
<td>Advanced Javascript</td>
<td>Javascript</td>
<td>Ex. 5— JavaScript Project</td>
</tr>
<tr>
<td>Exam Rehersal</td>
<td>Exam Rehersal</td>
<td>Javascript</td>
<td></td>
</tr>
</tbody>
</table>
Summary

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