Why study PL? (discussion)

“A different language is a different vision of life”
- Fellini

- Hypothesis:
  Programming language shapes programming thought

- Characteristics of a language affect how ideas can be expressed in the language

Course Goals

“Free your mind”
- Morpheus

You will learn several new
- languages and constructs
- ways to describe and organize computation

Yes, you can do that in Java/Assembly but ...
So what does studying PL buy me?

Enables you to create software that is

- Readable
- Correct
- Extendable
- Modifiable
- Reusable

So what does studying PL buy me?

Will help you learn new languages

- There was no Java (C#) 13 (8) years ago
- Will learn the anatomy of a PL
- Fundamental building blocks of languages reappear in different guises in different languages and different settings
- Re-learn the languages you already know

So what does studying PL buy me?

Enables you to design new languages

“who, me?”

Buried inside any extensible system is a PL
- Emacs: E-Lisp
- Word, Powerpoint: VBScript
- Quake: QuakeC
- SQL, Renderman, LaTeX, XML...

So what does studying PL buy me?

Enables you to better choose the right language

“But isn’t that decided by
- libraries,
- standards,
- and my boss?”

Yes. Chicken-and-egg.

My goal: educate tomorrow’s tech leaders & bosses
So you’ll make considered, informed choices
So what does studying PL buy me?

Makes you look at things in different ways, think outside of the box

Knowing language paradigms other than traditional ones will give you new tools to approach problems, even if you are programming in Java

PL Dimensions (discussion)

Dimension: Syntax

- Languages have different syntax
  - But the difference in syntax can be superficial
  - C# and Java have different syntax, but are very similar
- In this class, will look beyond superficial syntax to understand the underlying principles
**Dimension: Computation model**

- Functional: Lisp, OCaml, ML
- Imperative: Fortran, Pascal, C
- Object oriented: Smalltalk, C++, Java, C#
- Constraint-based: Prolog, CLP(R)

**Dimension: Memory model**

- Explicit allocation-deallocation: C, C++
- Garbage collection: Smalltalk, Java, C#
- Regions: safe versions of C (e.g. Cyclone)
  - allocate in a region, deallocate entire region at once
  - more efficient than GC, but no dangling ptrs

**Dimension: Typing model**

- Statically typed: Java, C, C++, C#
- Dynamically typed: Lisp, Scheme, Perl, Smalltalk
- Strongly typed (Java) vs. weakly typed (C, C++)

**Dimension: Scoping model**

- What x gets read inside of f?
- Static scoping says the global x
- Dynamic scoping says the one from g

```plaintext
int x;
int x;
f(){
  ... x ...
}
g(int x) {
  f();
}
```
Dimension: Execution model

- Compiled: C, C++
- Interpreted: Perl, shell scripting PLs
- Hybrid: Java

- Is this really a property of the language? Or the language implementation?
- Depends...

So many dimensions

- Yikes, there are so many dimensions!
- How to study all this!
- One option: study each dimension in turn
- In this course: explore the various dimensions by looking at a handful of PLs

Course material

Outline:
1. Functional, OCaml, 4 weeks
2. Modular, Java, 1 week
3. OO, Python, 3 weeks
4. Logic, Prolog, 1 weeks

Recommended Text: {1,4}
Modern Programming Languages, Adam Webber
Not very used in this class - pay attention to lecture

Resources for {2,3} will be posted on webpage

Requirements and Grading

- Midterm (open book): 30%
- Prog. Assignments (6-8): 35%
- Final (cheat sheet allowed): 35%
Programming Assignments

- Unfamiliar languages and environments:  
  - start early!
- Done alone, unless otherwise specified  
  - Please see academic integrity statement
- Scoring: Style + Test suite
- Forget Java/C: Immerse yourself in new PL
- Assignment #1 on web. Due Jan 16, 11:59:00pm

When we meet

- Lectures: Sorin Lerner, Tu-Th 12:30-1:50pm in SEQUO 147
- Section: Ravi Chugh, ???

All this info, plus much more, on the web:  
[www.cs.ucsd.edu/classes/wi09/cse130/](http://www.cs.ucsd.edu/classes/wi09/cse130/)

Enough with the small talk

Say hello to OCaml

```c
void sort(int arr[], int beg, int end){
if (end > beg + 1){
int piv = arr[beg];
int l = beg + 1;int r = end;
while (l != r-1){
    if(arr[l] <= piv)
        l++;
    else
        swap(&arr[l], &arr[r--]);
}
if(arr[l]<piv && arr[r]<piv)  
l=r+1;
else if(arr[l]<piv && arr[r]>piv) 
{l++; r--};
else if (arr[l]>piv && arr[r]<piv)
    swap(&arr[l++], &arr[r--]);
else
    r=l-1;
    swap(&arr[--r], &arr[beg]);
sort(arr, beg, r);
sort(arr, l, end);
}
}
```

```ocaml
let rec sort l = 
match l with [] -> []
|(h::t) -> 
    let(l,r)= List.partition ((<=) h) t in 
    (sort l)@h::(sort r)

Quicksort in C

Quicksort in Ocaml
```
Why readability matters...

Quicksort in J

\[
\text{sort=}:: (\$:@(<#[],(=#[]),\$:@(>#[]))(\sim \ ?@#)) ^: (1:<#)
\]

Quicksort in OCaml

let rec sort l =
  match l with
  | [] -> []
  | (h::t) ->
    let (l,r) = List.partition ((<=) h) t in
    (t @ h @ t) @ h @ (t @ h @ r)

Say hello to OCaml

Plan (next 4 weeks)

1. Fast forward
   • Rapid introduction to what’s in OCaml

2. Rewind

3. Slow motion
   • Go over the pieces individually

History, Variants

“Meta Language”
• Designed by Robin Milner @ Edinburgh
• Language to manipulate Theorems/Proofs
• Several dialects:
  - Standard” ML (of New Jersey)
    • Original syntax
  - “O’Caml: The PL for the discerning hacker”
    • French dialect with support for objects
    • State-of-the-art
    • Extensive library, tool, user support
    • (.NET)
ML’s holy trinity

- Everything is an expression
- Everything has a value
- Everything has a type

Interacting with ML

“Read-Eval-Print” Loop

Repeat:
1. System reads expression \( e \)
2. System evaluates \( e \) to get value \( v \)
3. System prints value \( v \) and type \( t \)

What are these expressions, values and types?

Base type: Integers

\[
\begin{align*}
2 &\rightarrow 2 \\
2+2 &\rightarrow 4 \\
2 \times (9+10) &\rightarrow 38 \\
2 \times (9+10) -12 &\rightarrow 26 \\
\end{align*}
\]

Type: \textit{int}

Complex expressions using “operators”:\textit{(why the quotes?)}
- \(+\), \(-\), \(*\)
- \(\text{div}\), \(\text{mod}\)

Base type: Strings

\[
\begin{align*}
"ab" &\rightarrow "ab" \\
"ab" ^ "xy" &\rightarrow "abxy" \\
\end{align*}
\]

Type: \textit{string}

Complex expressions using “operators”:\textit{(why the quotes?)}
- Concatenation \(^\)
Base type: Booleans

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>1 &lt; 2</td>
<td>true</td>
</tr>
<tr>
<td>&quot;aa&quot; = &quot;pq&quot;</td>
<td>false</td>
</tr>
<tr>
<td>(&quot;aa&quot; = &quot;pq&quot;) &amp;&amp; (1&lt;2)</td>
<td>false</td>
</tr>
<tr>
<td>(&quot;aa&quot; = &quot;pq&quot;)</td>
<td></td>
</tr>
</tbody>
</table>

Complex expressions using “operators”:
- “Relations”: =, <, <=, >
- &&, ||, not

Type Errors

- Untypable expression is rejected
  - No casting or coercing
  - Fancy algorithm to catch errors
  - ML’s single most powerful feature

Complex types: Product (tuples)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2+2, 7&gt;8)</td>
<td>int * bool</td>
</tr>
<tr>
<td>(9-3, &quot;ab&quot;^&quot;cd&quot;, (2+2, 7&gt;8))</td>
<td>int * string * (int * bool)</td>
</tr>
<tr>
<td>(6, &quot;abcd&quot;, (4,false))</td>
<td></td>
</tr>
</tbody>
</table>

• Triples,…
• Nesting:
  - Everything is an expression, nest tuples in tuples
## Complex types: Lists

- Unbounded size
- Can have lists of anything
- But...

### List operator “Cons” ::
- `1::[];`  →  `1::[1];`  →  `1::[1;2];`
- `"a"::["b"; "c"];`  →  `"a"::["b"; "c"; "d"];`
- `[(1,"a";"b");(3+4,"c")];`  →  `[(1,"a";"b");(3+4,"c";"d")];`

Can only “cons” element to a list of **same type**
- `1::["b"; "cd"];`

### List operator “Append” @
- `[1;2]@[3;4;5];`  →  `1;2;3;4;5;`
- `["a"]@["b"];`  →  `"a";"b";`
- `[]@[1];`  →  `1;`

Can only append two lists **of the same type**
- `1 @[2;3];`
- `[1] @ ["a"; "b"];`
Complex types: Lists

List operator “head” `hd`

- `hd [1;2];` → `1` `int`
- `hd ([“a”]@[“b”]);` → “a” `string`

Only take the head a nonempty list `hd [];`

Complex types: Lists

List operator “tail” `tl`

- `tl [1;2;3];` → `[2;3]` `int list`
- `tl ([“a”]@[“b”]);` → [“b”] `string list`

Only take the tail of nonempty list `tl [];`

Recap: Tuples vs. Lists?

What’s the difference?

• Tuples:
  - Different types, but fixed number:
    - pair = 2 elts `(3, “abcd”)` `(int * string)`
    - triple = 3 elts `(3, “abcd”,(3.5,4.2))` `(int * string * (real * real))`

• Lists:
  - Same type, unbounded number:
    - `[3;4;5;6;7]` `int list`

• Syntax:
  - Tuples = comma
  - Lists = semicolon
So far, a fancy calculator...

... what do we need next?