If we ignore types, then ML let-bindings can be syntactic sugar for calling an anonymous function:

\[
\text{let } \text{val } x = e1 \text{ in } e2 \text{ end}
\]

\[
(\text{fn } x => e2) \ e1
\]

These both evaluate \( e1 \) to \( v1 \), then evaluate \( e2 \) in an environment extended to map \( x \) to \( v1 \).

So exactly the same evaluation of expressions and result.

But in ML, there is a type-system difference:

- \( x \) on the left can have a polymorphic type, but not on the right.
- Can always go from right to left.
- If \( x \) need not be polymorphic, can go from left to right.

---

**Course Motivation**

(Did you think I forgot? 😊)

- Why learn the fundamental concepts that appear in all (most?) languages?
- Why use languages quite different from C, C++, Java, Python?
- Why focus on functional programming?
- Why use ML, Racket, and Ruby in particular?
- Not: Language X is better than Language Y

[You won’t be tested on this stuff]
More on cars

• A good mechanic might have a specialty, but also understands how “cars” (not a particular make/model) work
  – The upholstery color isn’t essential (syntax)
• A good mechanical engineer really knows how cars work, how to get the most out of them, and how to design better ones
  – I don’t have a favorite kind of car or a favorite PL
• To learn how car pieces interact, it may make sense to start with a classic design rather than the latest model
  – A popular car may not be best
  – May especially not be best for learning how cars work

Why semantics and idioms

This course focuses as much as it can on semantics and idioms

• Correct reasoning about programs, interfaces, and compilers requires a precise knowledge of semantics
  – Not “I feel that conditional expressions might work like this”
  – Not “I like curly braces more than parentheses”
  – Much of software development is designing precise interfaces, what a PL means is a really good example
• Idioms make you a better programmer
  – Best to see in multiple settings, including where they shine
  – See Java in a clearer light even if I never show you Java

Hamlet

The play Hamlet:
  – Is a beautiful work of art
  – Teaches deep, eternal truths
  – Is the source of some well-known sayings
  – Makes you a better person

Continues to be studied centuries later even though:
  – The syntax is really annoying to many
  – There are more popular movies with some of the same lessons
  – Reading Hamlet will not get you a summer internship

All cars are the same

• To make it easier to rent cars, it is great that they all have steering wheels, brakes, windows, headlights, etc.
  – Yet it is still uncomfortable to learn a new one
  – Can you be a great driver if you only ever drive one car?
• And maybe PLs are more like cars, trucks, boats, and bikes
• So are all PLs really the same...

Are all languages the same?

Yes:
  – Any input-output behavior implementable in language X is implementable in language Y (Church-Turing thesis)
  – Java, ML, and a language with one loop and three infinitely-large integers are “the same”
Yes:
  – Same fundamentals reappear: variables, abstraction, one-of types, recursive definitions, ...
No:
  – The human condition vs. different cultures (travel to learn more about home)
  – The primitive/default in one language is awkward in another
  – Beware “the Turing tarpit”

Functional Programming

Why spend 60-80% of course using functional languages:
  – Mutation is discouraged
  – Higher-order functions are very convenient
  – One-of types via constructs like datatypes

Because:
1. These features are invaluable for correct, elegant, efficient software (great way to think about computation)
2. Functional languages have always been ahead of their time
3. Functional languages well-suited to where computing is going

Most of course is on (1), so a few minutes on (2) and (3) …
Ahead of their time

All these were dismissed as "beautiful, worthless, slow things PL professors make you learn"
- Garbage collection (Java didn’t exist in 1995, PL courses did)
- Generics (List<T> in Java, C#), much more like SML than C++
- XML for universal data representation (like Racket/Scheme/LISP/…)
- Higher-order functions (Ruby, Javascript, C#, now Java, …)
- Type inference (C#, Scala, …)
- Recursion (a big fight in 1960 about this – I’m told 😊)
- …

The future may resemble the past

Somehow nobody notices we are right… 20 years later
- "To conquer" versus “to assimilate”
- Societal progress takes time and muddles “taking credit”
- Maybe pattern-matching, currying, hygienic macros, etc. will be next

Recent-ish Surge, Part 1

Other popular functional PLs (alphabetized, pardon omissions)
- Clojure http://clojure.org
- Erlang http://www.erlang.org
- F# http://tryfsharp.org
- Haskell http://www.haskell.org
- OCaml http://ocaml.org
- Scala http://www.scala-lang.org

Some “industry users” lists (surely more exist):
- http://www.haskell.org/haskellwiki/Haskell_in_industry
- http://ocaml.org/companies.html
- In general, see http://cufp.org

Recent-ish Surge, Part 2

Popular adoption of concepts:
- C#, LINQ, C++ (closures, type inference, …)
- Java 8 (closures)
- MapReduce / Hadoop
- Avoiding side-effects essential for fault-tolerance here
- Scala libraries (e.g., Akka, …)
- …

Why a surge?

My best guesses:
- Concise, elegant, productive programming
- JavaScript, Python, Ruby helped break the Java/C/C++ hegemony
- Avoiding mutation is the easiest way to make concurrent and parallel programming easier
- In general, to handle sharing in complex systems
- Sure, functional programming is still a small niche, but there is so much software in the world today even niches have room

The languages together

SML, Racket, and Ruby are a useful combination for us

<table>
<thead>
<tr>
<th>Functional</th>
<th>dynamically typed</th>
<th>statically typed</th>
</tr>
</thead>
<tbody>
<tr>
<td>SML</td>
<td>Racket</td>
<td>Java</td>
</tr>
<tr>
<td>object-oriented</td>
<td>Ruby</td>
<td></td>
</tr>
</tbody>
</table>

ML; polymorphic types, pattern-matching, abstract types & modules
Racket: dynamic typing, “good” macros, minimalist syntax, eval Ruby: classes but not types, very OOP-mimes [and much more]

Really wish we had more time:
- Haskell: laziness, purity, type classes, monads
- Prolog: unification and backtracking [and much more]
But why not…

Instead of SML, could use similar languages easy to learn after:
- OCaml: yes indeed but would have to port all my materials 😔
  - And a few small things (e.g., second-class constructors)
- F#: yes and very cool, but needs a .Net platform
  - And a few more small things (e.g., second-class constructions, less elegant signature-matching)
- Haskell: more popular, cooler types, but lazy semantics and type classes from day 1
- Coq: truly immutable, even crazier types, and even proofs!

Admittedly, SML and its implementations are showing their age (e.g., and also less tool support), but it still makes for a fine foundation in statically typed, eager functional programming.

But why not…

Instead of Racket, could use similar languages easy to learn after:
- Scheme, Lisp, Clojure, Python…
  - Racket has a combination of:
    - A modern feel and active evolution
    - “Better” macros, modules, structs, contracts…
    - A large user base and community (not just for education)
    - An IDE tailored to education

Could easily define our own language in the Racket system
- Would rather use a good and vetted design

But why not…

Instead of Ruby, could use another language:
- Python, Perl, JavaScript are also dynamically typed, but are not as “fully” OOP, which is what I want to focus on
  - Python also does not have (full) closures
  - JavaScript also does not have classes but is OOP
- Smalltalk serves my OOP needs
  - But implementations merge language/environment
    - Less modern syntax, user base, etc.

Is this real programming?

- The way we use ML/Racket/Ruby can make them seem almost “silly” precisely because lecture and homework focus on interesting language constructs.
- “Real” programming needs file I/O, string operations, floating-point, graphics, project managers, testing frameworks, threads, build systems…
  - Many elegant languages have all that and more
    - Including Racket and Ruby
  - If we used Java the same way, Java would seem “silly” too

A note on reality

Reasonable questions when deciding to use/learn a language:
- What libraries are available for reuse?
- What tools are available?
- What can get me a job?
- What does my boss tell me to do?
- What is the de facto industry standard?
- What do I already know?

Our course by design does not deal with these questions
- You have the rest of your life for that
- And technology leaders affect the answers