CSE 341:
Programming Languages

Hal Perkins
Spring 2011
Lecture 6— Nested pattern-matching; course motivation
Patterns

What we know:

- case-expressions do pattern-matching to choose branch
- val-bindings and fun-arguments also do pattern-matching
  - All functions take one argument
- Can match datatypes (including lists, options) and records (including tuples)

The full story is *more general* — patterns are much richer than we have let on.
Deep patterns

The full definition of pattern-matching is recursive, processing the matched-on value and the pattern together.

A pattern can be:

- A variable (matches everything, introduces a binding)
- _ (matches everything, no binding)
- A constructor C (matches value C, if C carries no data)
- A constructor and a pattern (e.g., C p) (matches a value if the value “is a C” and p matches the value it carries)
- A pair of patterns ((p₁, p₂)) (matches a pair if p₁ matches the first component and p₂ matches the second component)
- A record pattern...
- ...

Hal Perkins CSE341 Spring 2011, Lecture 6 3
Can you handle the truth?

It's really:

- \texttt{case } e \texttt{ of } p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n \\
- \texttt{val } p = e \\
- \texttt{fun } f \ p_1 = e_1 \mid f \ p_2 = e_2 \ldots \mid f \ p_n = e_n \\

Inexhaustive matches may raise exceptions and are bad style.

Example: could write pattern \texttt{Add } \mathtt{pr} \texttt{ or } \texttt{Add } (e_1, e_2) \\

Again: The definition of pattern-matching is recursive over the value-being-matched and the pattern.

_ and binding a variable are just base cases.
Some function examples

- fun f1 () = 34
- fun f2 _ = 34
- fun f3 (x,y) = x + y
- fun f4 pr = let val (x,y) = pr in x + y end

Is there any difference to callers between f3 and f4?

In most languages, “argument lists” are syntactically separate, second-class constructs.

Can be useful: f3 (if e1 then (3,2) else pr)

- (We discussed this on Wednesday too.)

See lec6.sml for a few examples where nested patterns are quite nice.
Course Motivation

I owe you an answer to why 341 has material worth learning.

1. Why learn programming languages that are quite different from Java, C, C++?

2. Why learn the fundamental concepts that appear in all (most?) programming languages?

3. Why focus on functional programming (avoiding mutation, embracing recursion, and writing functions that take/return other functions)?
A couple questions...

What’s the best car?

What are the best kind of shoes?

What is the correct house?
Aren’t all languages the same?

Yes: Any input-output behavior you can program in language X you can program in language Y

- Java, ML, and a language with one loop and three infinitely-large integers are “equal”
- This is called the “Turing tarpit”

Yes: Certain fundamentals appear in most languages (variables, abstraction, one-of types, *recursive definitions*, …)

- Travel to learn more about where you’re from
- ML, Scheme, Ruby well-suited for letting these fundamentals shine

No: Most cars have 4 tires, 2 headlights, …

- Mechanics learn general principles and what’s different
Aren’t the semantics my least concern?

Admittedly, there are many important considerations:

- What libraries are available?
- What can get me a summer internship?
- What does my boss tell me to do?
- What is the de facto industry standard?
- What do I already know?

Technology leaders affect the answers to these questions.

Sound reasoning about programs, interfaces, and compilers requires knowledge of semantics.

And there is a place in universities for learning deep truths and beautiful insights as an end in itself. (Like watching Hamlet.)
Aren’t languages somebody else’s problem?

If you design an extensible software system, you’ll end up designing a (small?) programming language!

Examples: VBScript, JavaScript, PHP, ASP, QuakeC, Renderman, bash, AppleScript, emacs, Eclipse, AutoCAD, ...
Functional programming

Okay, so why ML and Scheme where:

- Mutation is discouraged
- Datatype-based one-of types
- Higher-order functions (next week)

Because:

1. These features are invaluable for correct, elegant, efficient software (great way to think about computation).
2. Functional languages have a history of being ahead of their time
3. They are well-suited to where computing is going (multicore and data centers)

Much of the course is (1), so let’s give an infomercial for (2) and (3)…
Ahead of their time

- Garbage collection (Java didn’t exist in 1995, SML & Scheme did)
- Generics (List<T> in Java, C#), much more like SML than C++
- XML for universal data representation (like Scheme / Lisp)
- Function closures in Python, Ruby, etc.
- Ruby’s iterators lifted from CLU (another “useless language”)
- ...

All features dismissed as, “fine for academics, but will never make it in the real world”.

- Maybe datatypes or currying or multimethods will be next...
- “Conquering” vs. “assimilation”
Recent Surge

- F#, C#, LINQ, Scala, Java 8
- Multicore computing (no mutation = easier to parallelize)
- MapReduce / Hadoop (first published in 2004)
- Small companies (Jane Street, Galois, many others)
  - And not so small (Ericsson’s Erlang)
  - All consider functional programming a key competitive advantage
    * In part for hiring smarter people
- Lots of research projects

Note: None of these examples use SML or Scheme, but that’s okay: think how much you’ve learned in the last 10 days. They are all informed and influenced by these ideas.
Summary

There is no such thing as a “best programming language”. (There are good general design principles we will study.)

A good language is a relevant, crisp, and clear interface for writing software.

Software leaders should know about programming languages.

Learning languages has super-linear payoff.

• But you have to learn the semantics and idioms, not a cute syntactic trick for printing “Hello World”.

Functional languages have been on the leading edge for decades, but ideas get absorbed by the masses slowly.

• Perhaps things are starting to change?

• Even if not, it will make you a better Java/C programmer