CSE 341: Programming Languages

Dan Grossman

Fall 2004

Lecture 6— The truth about bindings and course motivation

Where we are

Class has covered tremendous ground—you should catch up by doing homework 2.

Next time we'll take up first-class functions (closures, functions as values):

- A really key idea in computer science

But we haven't yet seen that pattern-matching is an elegant generalization of variable binding.

And I owe you an explanation of why we should study programming languages, particulary ML, Scheme, and Smalltalk

Deep patterns

Patterns are much richer than we have let on. A pattern can be:

- A variable (matches everything, introduces a binding)
- _ (matches everything, no binding)
- A constructor and a pattern (e.g., `C` $\boldsymbol{p}$) (matches a value if the value "is a `C`" and $\boldsymbol{p}$ matches the value it carries)
- A pair of patterns (`(`$\boldsymbol{p1}$`,`$\boldsymbol{p2}$`)`) (matches a pair if $\boldsymbol{p1}$ matches the first component and $\boldsymbol{p2}$ matches the second component)
- A record pattern...
- An integer constant...
- ...

Can you handle the truth?

It's really:

- `val p = e`
- `fun f p1 = e1 | f p2 = e2 ... | f pn = en`
- `case e of p1 => e1 | ... | pn => en`

Inexhaustive matches may raise exceptions and are bad style.

Example: could write `Rope pr` or `Rope (r1,r2)`

Fact: Every ML function takes exactly one argument!

Some function examples

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

Is there *any* difference to callers between `f2` and `f3`?

In most languages, "argument lists" are syntactically separate, *second-class* constructs.

Can be useful: `f2 (if e1 then (3,2) else pr)`

A question?

What's the best car?

What are the best kind of shoes?

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, each-of types, *inductive definitions*, ...)

- Travel to learn more about where you're from

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

- Mechanics learn general principles and what's different

Aren't these academic languages worthless?

In the short-term, maybe: Not many summer internships using ML?

But:

- Knowing them makes you a better Java, C, and Perl programmers (affects your idioms)
- Java did not exist in 1993; what does not exist now?
- Do Java and Scheme have anything in common? (Hint: check the authors)
- Eventual vindication: garbage-collection and generics

Aren't the semantics my least concern?

Admittedly, there are many important considerations:

- What libraries are available?
- 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.

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, ...

Another view: A language is an interface with just a few functions (evaluate, typecheck) and a sophisticated input type.

In other words, an interface is just a stupid programming language.

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".

End of the course: Language-design goals, mechanisms, and trade-offs

Next time: why ML, Scheme, and Smalltalk?