CSE 341:
Programming Languages

Winter 2005

Lecture 7— Motivation and First-Class Functions

Today

- Finish course motivaton
- Summarize what we've learned with a concise and well-known notation for recursively-defined language constructs
- Begin first-class functions

Why these 3?

Functional programming (ML, Scheme) encourages recursion, discourages mutation, provides elegant, lightweight support for first-class code. Support for extensibility complements OO.

- ML has a polymorphic type system (vindication imminent!) complementary to OO-style subtyping, a rich module system for abstract types, and rich pattern-matching.
- Scheme has dynamic typing, “good” macros, fascinating control operators, and a minimalist design.
- Smalltalk has classes but not types, an unconventional environment, and a complete commitment to OO.

Runners-up: Haskell (laziness and purity), Prolog (unification and backtracking), thousands of others...

Why not some popular ones?

- Java: you know it, will contrast at end of course (e.g., interfaces, anonymous inner classes, container types)
- C: lots of “implementation-dependent” behavior (a bad property), and we have CSE303
- C++: an enormous language, and unsafe like C
- Perl: advantages (strings, files, ...) not foci of this course. Python or Ruby would be closer.

Are these useful?

The way we use ML/Scheme/Smalltalk in 341 can make them seem almost "silly" precisely because we focus on *interesting language concepts*

"Real" programming needs file I/O, strings, floating-point, graphics libraries, project managers, unit testers, threads, foreign-function interfaces, ...

- These languages have all that and more!
- If Java were in 341, it would seem "silly" too

Somewhat outdated links:

- ML: http://www.cs.princeton.edu/~appel/smlnj/projects.html
- OCaml: http://caml.inria.fr/users_programs-eng.html

Summary and Some Notation

Learned the syntax, typing rules, and semantics for (a big) part of ML

Can summarize *abstract syntax* with *(E)BNF*. Informally:

t ::= `int` | `bool` | `unit` | `dtname`
  | t_1 `->` t_2 | t_1 `*` t_2 | `{x1=`t_1`, ..., xn=`t_n`}`
e ::= `34` | `x` | `(`e_1`,`e_2`)` | `if` e_1 `then` e_2 `else` e_3
  | `let` b_1 `...` b_n `in` e `end` | e_1 e_2
  | `case` e `of` p_1 `=>` e_1 `| ... |` p_n `=>` e_n
  | e_1 `+` e_2 | `{x1=`e_1`, ..., xn=`e_n`}` | `C` e
b ::= `val` p `=` e | `fun f` p `=` e
  | `datatype dtname = C1 of` t_1 `| ... | Cn of` t_n
p ::= `34` | `x` | `_` | `C` p | `(`p_1`,`p_2`)` | `{x1=`p_1`, ..., xn=`p_n`}`

Things left out of this *grammar*: n-tuples, field-accessors, floating-point, boolean constants, andalso/orelse, lists, ...

First-Class Functions

- Functions are values. (Variables in the environment are bound to them.)
- We can pass functions to other functions.
  - *Factor* common parts and *abstract* different parts.
- Most polymorphic functions take functions as arguments.
  - Non-example: `fun f x = 42`
- Some functions taking functions are polymorphic.

Type Inference and Polymorphism

ML can infer function types based on function bodies. Possibilities:

- The argument/result must be one specific type.
- The argument/result can be *any* type, but may have to be the *same type* as other parts of argument/result.
- Some hand-waving about "equality types"

We will study this *parametric polymorphism* more later.

Without it, ML would be a pain (e.g., a different list library for every list-element type).

Fascinating: If `f:int->int`, there are lots of values `f` could return. If `f:'a->'a`, whenever `f` returns, it returns its argument!

Anonymous Functions

As usual, we can write functions anywhere we write expressions.

- We already could:

  ```
  (let fun f x = e in f end)
  ```

- Here is a more concise way (better style when possible):

  ```
  (fn x => e)
  ```

- Cannot do this for recursive functions (why?)

Returning Functions

Syntax note: `->` "associates to the right"

- `t1->t2->t3` means `t1->(t2->t3)`

Again, there is nothing new here.

The key question: What about *free variables* in a function value? What *environment* do we use to *evaluate* them?

Are such free variables useful?

You must understand the answers to move beyond being a novice programmer.