Functions and Closures

March 11, 2002

CS 131: Programming Languages

Code Duplication

- Often tempting to duplicate useful pieces of code in a program (possibly with minor changes).
  - Or, several independent implementations of the same task
- This has serious disadvantages (why?)

Code Duplication

"We heard firsthand of a U.S. state whose governmental computer systems were surveyed for Y2K compliance. The audit turned up more than 10,000 programs, each containing its own version of Social Security number validation."

Hunt and Thomas, *The Pragmatic Programmer*

Avoiding Duplication

- Many language mechanisms are designed to avoid the need for duplication
  - i.e., to permit code reuse
- Such as?

Functions/Procedures

- Fairly obvious that functions (and procedures and subroutines) permit code reuse.
- So today, we'll just revisit higher-order functions.
  - Abstracting control flow
  - Closures
    - first-class functions as values

Abstracting Control Flow

- Given a data structure, certain patterns of control tend to re-occur
- E.g., lists
  - Do something to every element of a list
  - Apply a transformation to every element of a list.
  - Process each list element, updating an accumulator

List Transformations

```
fun double_list ([] : int list) = []
  | double_list (n::ns) =
        (2*n) :: (double_list ns)

fun square_list ([] : real list) = []
  | square_list (r::rs) =
        (r*r) :: (square_list rs)

fun asciify_list ([] : char list) = []
  | asciify_list (c::cs) =
        (Char.ord c) :: (asciify_list cs)
```

Better List Transformations

```
fun map f [] = []
  | map f (x::xs) = (f x) :: (map f xs)

val double_list = map (fn n => 2*n)

val square_list = map (fn (r:real) => r*r)

val asciify_list = map Char.ord
```

Complex Control Flow

- First-class functions can also be used to implement complicated control-flow in a program
- Examples:
  - Backtracking search (regular expressions)
  - Aborting computation

Problem 1: Regular Expressions

```
datatype regexp = Zero    (* Empty Language *)
                | Epsilon (* Accepts empty string *)
                | Char  of char
                | Plus  of regexp * regexp
                | Times of regexp * regexp
                | Star  of regexp

val accept : regexp * char list -> bool
```

Problem 2: List Multiplications

- Given a list of integers, write a function that returns their product.