CS 42—Structural recursion and higher-order functions (HOFs)

Tuesday, October 9, 2018

Summary

Today, we'll explore recursive algorithms over inductive data structures (specifically, linked lists); and we'll learn how write functions that can be used as input to and output from other functions.

Terminology

higher-order function: a function that takes another function as an argument or that returns a function as its result (or both).

anonymous function: a function without a name, also referred to as a "lambda".

Notation for describing types

Here is some notation that we'll use to describe types. You won't be tested on this notation; it's just a useful shorthand that helps us describe the code we write.

Notation	Meaning
`Int`, `Bool`, `Char`, `String`, etc.	A "primitive" type, i.e., one that's built into Racket and is not a list or a function.
A → *B*	A function that takes a parameter of type A and results in a value of type B. For example the `not` function has type `Bool → Bool`.
A x *B* → *C*	A function that takes one parameter of type A, one parameter of type B, and results in a value of type C. For example the + function has type `Int x Int → Int`.
[*A*]	A list of type A, for example the value `'(1 2 3)` has type `[Int]`.

Recap: Lists in Racket

Racket lists are linked lists, where each "link" in the list is a pair of element + list.

Operation	Meaning
List-building operations	
`empty` *or* `'()`	constructs an empty list
(cons <*value*> <*list*>)	constructs a new list by prepending an element to an existing list (**Careful!** If the second element is not a list, then `cons` does not construct a new list.)
(list <*value*$_1$> ... <*value*$_N$>)	constructs a list with the given arguments as elements
'(<*value*$_1$> ... <*value*$_N$>)	
(append <*list*$_1$> ... <*list*$_N$>)	append multiple lists into a single list
List-accessing operations	
`(empty? <value>)`	returns true if the argument is an empty list
`(first <list>)`	returns the head of a non-empty list
`(rest <list>)`	returns the tail of a non-empty list

Common higher-order functions (HOFs)

Here are some common higher-order functions that come with Racket. These functions correspond to common patterns in programming.

`(map f L)`: given a transforming function `f` and a list `L`, `map` produces a new list `L'` where each element of `L'` is the result of applying `f` to the corresponding element of `L`. The function `f` takes a single element of `L` and transforms it to a new value. `map` has type `(A → B) x [A] → [B]`.

`(filter f L)`: given a predicate function `f` and a list `L`, `filter` produces a new list `L'` that contains *only* the elements of `L` for which the predicate is `true`. The function `f` takes a single element of `L` and returns either `true` or `false`. `filter` has type `(A → Bool) x [A] → [A]`.

`(foldl f seed L)`: given a folding function `f`, an initial `seed` value, and a list `L`, `foldl` reduces `L` to a single value by repeatedly applying `f` to the list. The function `f` takes *two* arguments: an element of `L` and the accumulated value; it returns a new accumulated value. `foldl` starts by applying `f` to the *first* element of `L` and the `seed`, then moves its way towards the *end* of the list:

$$\texttt{(foldl f seed L)} \equiv \texttt{(f } v_n \texttt{ (… (f } v_1 \texttt{ (f } v_0 \texttt{ seed)) …))}$$

`(foldr f seed L)`: like `foldl` except it starts by applying `f` to the *last* element of `L` and the `seed`, then moves its way towards the *front* of the list:

$$\texttt{(foldr f seed L)} \equiv \texttt{(f } v_0 \texttt{ (… (f } v_{n-1} \texttt{ (f } v_n \texttt{ seed)) …))}$$

`foldl` and `foldl` have type `(A x B → B) x B x [A] → B`.

Caution: when `f` is not associative (e.g., when `f` is subtraction), `foldr` and `foldl` can return different results.

anonymous functions (i.e., lambdas)

The result of evaluating this expression is a value whose type is a function:

*(**lambda** (parameter$_1$ … parameter$_n$) body-expr)*

For example, the expression `(lambda (x y) (+ x y))` has type `Int x Int → Int`.

It may be helpful to think of lambda as the most basic way of defining a function, so that this:

```
(define (f x y)
  (+ x y))
```

is just syntactic sugar for this:

```
(define f (lambda (x y) (+ x y)))
```

We often use anonymous functions when we want to pass an argument or to return a value whose type is a function, for example: `(map (lambda (x) (* x 2)) '(1 2 3))`

Next time: Branching recursion ("use it or lose it") and analysis of programs.