CS 42—Introduction to Functional Programming & Racket

Tuesday, October 2, 2018

Summary

Today, we will introduce the concept of functional programming and discuss why we're studying it. We'll also explore a functional programming language called Racket.

Terminology

primitive value: In a programming language, a primitive value is a literal value that's built into the language. Examples from Racket include `42` (an integer), `42.0` (a floating point number), and `"platypus"` (a string).

primitive operation: In a programming language, a primitive operation is a function that's built into the language. Examples from Racket include `+` (addition) and `not` (logical negation).

evaluate: In Racket, "evaluate" means "reduce to a primitive value".

expression: An expression is something that can be evaluated. For example 3 + 4 can be evaluated to produce 7.

subexpression: A subexpression is an expression within another expression. For example, in the expression 7 * (3 + 4), both 7 and (3 + 4) are subexpressions.

s-expression: short for "symbolic expression". In Racket, every expression is an s-expression, which is either a primitive value or has the form `(`*op* arg_1 arg_2 … arg_n`)`, where *op* and arg_i can themselves be s-expressions. The first subexpression *op* is treated as a function, and the remaining expressions arg_i are treated as arguments to the function.

side effect: In programming, a side effect of an expression is something the expression does to changes state in a way that persists even after the expression has been evaluated. Examples include: modifying a global variable, writing to the screen, and reading from the keyboard.

referential transparency: An expression has referential transparency if it always evaluates to the same result, given the same input. (Think: functions in math: 2^2 is always 4.)

Functional programming

From "Why Functional Programming Matters" by John Hughes (emphasis mine):

> *The special characteristics and advantages of functional programming are often summed up more or less as follows. Functional programs contain* ***no assignment statements****, so variables, once given a value, never change. More generally, functional programs contain* ***no side-effects*** *at all. A function call can have no effect other than to compute its result. This eliminates a major source of bugs, and also makes the order of execution irrelevant—since no side-effect can change an expression's value, it can be evaluated at any time. This relieves the programmer of the burden of prescribing the flow of control. Since expressions can be evaluated at any time, one can freely replace variables by their values and vice versa—that is, programs are "referentially transparent". This freedom helps make functional programs more tractable mathematically than their conventional counterparts.*

Functional programming in Racket

Here are some (though not all) of the primitive values and operations in Racket.

Type of primitive value	Example(s)
Integers	`42, -5`
booleans	`false, true`
floating-point numbers	`3.14`
rational numbers	$\frac{1}{5}$
characters	`#\a, #\b, #\c`
strings	`"platypus"`
symbols	`'x`

Primitive operation	Meaning
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`quotient`	integer division
`modulo`	remainder after integer division
`and`	logical and
`=`	numeric equality
`equal?`	general equality of values
`>`	greater than

Global constants

We use `define` to "bind" a global name to a value in our program (we won't use this very much).

```
(define name expr)
```

"Variables"

They're called variables, but we won't vary them (i.e., their values are constant).

```
(let* ([var1 expr1]
       [var2 expr2]
       …
       [varn exprn])
  body-expr)
```

Conditionals

If you have exactly one condition, use `if`; otherwise, use `cond` (which is like if / else if / else).

```
(if conditional-expr
    true-expr
    false-expr)
```

```
(cond [condition1 expr1]
       …
      [conditionn exprn]
      [else else-expr])
```

Functions

```
(define (function-name parameter1 … parametern)
  body-expr)
```

Next time: Lists & recursion in Racket