CS 313 Lecture 18

Functional Programming

Intro to Scheme
Part III: Functional Programming

Languages:
• Scheme
  • similar to LISP
• ML
  • typed functional language
• [Haskell]
Functional Programming

- motivated by problems of imperative languages which reflect the underlying von Neumann architecture
  - CPU interacts with Memory
    
    ```
    a := b + c;
    ```

- variables change state
- order of execution matters
- hard to parallelize
Referential Transparency

= meaning of a term is independent of context

• NOT the case in imperative languages
• e.g. the expression \((x + 1)\) changes meaning

\[
a = x + 1 \\
x = 5 \\
b = x + 1 \\
x = x + 1
\]
Referential Transparency

is the case in mathematics:

\[
\begin{align*}
a &= x + 1 \\
b &= x + 1 \\
\Rightarrow a &= b
\end{align*}
\]

assignment vs. equality

\[x = x + 1\]
Compiler optimization

• Common subexpression elimination:

\[ y = (2 \times a \times x + b) \times (2 \times a \times x + b); \]

Optimize:

\[ t = 2 \times a \times x + b; \]
\[ y = t \times t; \]
Compiler optimization

• What about
  \[ y = f(x) + f(x) \]
  Simplify to
  \[ y = 2 \times f(x) \]

• Can't do if \( f(x) \) has side-effects
  • changes global variables
  • prints to screen
  • saves to file
  • ...
Parallelism

• What about \( z = f(x) + g(y) \)
  Can we compute \( f(x), g(y) \) in parallel?

• Can't do if \( f(x), g(y) \) have *side-effects*
  • change global variables
  • print to screen
  • save to file
  • ...
*Pure* functional languages

- don't have side effects
- don't have assignment
- do have referential transparency
- are easy to parallelize

• Ok:
  
  \[
  \begin{align*}
  x & = 5 \\
  y & = 3 + x \\
  z & = x / y
  \end{align*}
  \]

• Not Ok:
  
  \[
  \begin{align*}
  x & = 5 \\
  x & = x + 2
  \end{align*}
  \]
Power of functional languages

• functions are first-class values
  • (just like numbers, lists, ...)
  • can pass to / return from functions

• simple:
  • focus on values, independent of storage allocation
    [need garbage collector]
Intro to Scheme  (Sethi Ch 10)

• introduced in 1975

• dialect of LISP (1958)
  • smaller, cleaner: easier to learn
  • true first-class functions
  • lexical scope
  • proper tail recursion

• will consider pure functional subset
  (though Scheme does allow assignment)
We'll program in Racket

- dialect of Scheme
- supports other languages / paradigms
- IDE: DrRacket
Scheme

• Uniform prefix syntax:

\[(\text{fun} \ arg1 \ arg2 \ \ldots)\]

\[
(+ 3 4) \quad ; \quad 3 + 4 \n\]

\[
(+ 3 4 5) \quad ; \quad 3 + 4 + 5 \n\]

\[
(+ 2 (* 3 5)) \quad ; \quad 2 + (3 * 5) \n\]

• bind names to values:

\[(\text{define a 5.5})\]

\[(\text{define b (+ 2 2)})\]
Functions

(define (fun-name params) expression)

(define (square x) (* x x))

(define (f x y) (+ (* x 2) y)) ; f(x,y) = 2x + y
Datatypes

• Booleans  
  \#t \#f

• Numbers  
  3  -7  2.0  2+3i  3/8

• Strings  
  "hello"  "how are you?"

• Symbols  
  'hello  'square

• Lists  
  '(hello (how are) you? 33 "hi")
Conditionals: if

(if expr then-result else-result)

(if (> a b) 3 4)

(define (max a b)
  (if (> a b) a b))

(define (sign a)
  (if (> a 0) 1
      (if (= a 0) 0 -1)))
Conditionals: cond

(cond (expr1 result1)

(expr2 result2)

...

(else result-n))

(define (sign a)

  (cond

    ((> a 0) 1)

    ((= a 0) 0)

    (else -1)))
Demo DrRacket