Today's topics
- Types of objects and procedures
- Procedural abstractions
- Capturing patterns across procedures – Higher Order Procedures

Types

(+ 5 10) ==> 15
(+ "hi" 5)

The object "hi", passed as the first argument to integer-add, is not the correct type

- Addition is not defined for strings

Types – simple data

- We want to collect a taxonomy of expression types:
  - Simple Data
    - Number
    - Integer
    - Real
    - Rational
    - String
    - Boolean
    - Names (symbols)

- We will use this for notational purposes, to reason about our code. Scheme does not directly check types of arguments as part of its processing.

Types – compound data

- Pair<A,B>
  - A compound data structure formed by a cons pair, in which the first element is of type A, and the second of type B. E.g. (cons 1 2) has type Pair<number, number>
- List<A>=Pair<A, List<A> or nil>
  - A compound data structure that is recursively defined as a pair, whose first element is of type A, and whose second element is either a list of type A or the empty list.
  - E.g. (list 1 2 3) has type List<number>; while (list 1 "string" 3) has type List<number or string>

Examples

25 ; Number
3.45 ; Number
"this is a string" ; String
(> a b) ; Boolean
(cons 1 3) ; Pair<number, number>
(list 1 2 3) ; List<number>
(cons "foo" (cons "bar" nil)) ; List<string>

Types – procedures

- Since procedures operate on objects, and return values, we can define their types as well.
- We will denote a procedures type by indicating the types of each of its arguments, and the type of the returned value, plus the symbol $\rightarrow$ to indicate that the arguments are mapped to the return value.
  - E.g. number $\rightarrow$ number specifies a procedure that takes a number as input, and returns a number as value
Types

• (+ 5 10) ==> 15
  (+ "hi" 5)

; The object "hi", passed as the first argument to integer-add, is not the correct type
• The type of the integer-add procedure is number, number ➔ number
  two arguments, both numbers
  result value of integer-add is a number
• Addition is not defined for strings

Type examples

• expression: evaluates to a value of type:
  15 number
  "hi" string
  square number ➔ number
  (> number,number ➔ boolean
  (> 5 4) ==> #t

• The type of a procedure is a contract:
  • If the operands have the specified types, the procedure will result in a value of the specified type
  • otherwise, its behavior is undefined
    • maybe an error, maybe random behavior

Types, precisely

• A type describes a set of scheme values
  • number ➔ number describes the set:
    all procedures, whose result is a number, that also require one argument that must be a number
• Every scheme value has a type
  • Some values can be described by multiple types
    • if so, choose the type which describes the largest set
  • Special form keywords like define do not name values
  • therefore special form keywords have no type

Your turn

• The following expressions evaluate to values of what type?

  (lambda (a b c) (if (> a 0) (+ b c) (- b c)))
  number, number, number ➔ number

  (lambda (p) (if p "hi" "bye"))
  Boolean ➔ string

  (* 3.14 (* 2 5))
  number

End of part 1

• type: a set of values
  • every value has a type
• procedure types (types which include ➔) indicate
  • number of arguments required
  • type of each argument
  • type of result of the procedure

• Types: a mathematical theory for reasoning efficiently about programs
  • useful for preventing certain common types of errors
  • basis for many analysis and optimization algorithms

What is procedure abstraction?

Capture a common pattern
  (* 2 2)
  (* 57 57)
  (* k k)

(lambda (x) (* x x))

Give it a name (define square (lambda (x) (* x x)))

Note the type: number ➔ number
Other common patterns

- $1 + 2 + \ldots + 100$
- $1 + 4 + 9 + \ldots + 100^2$
- $1 + 1/3^2 + 1/5^2 + \ldots + 1/101^2 \approx \pi/8$

(\texttt{define (sum-integers a b)}
  (if (> a b)
   0
   (+ a (\texttt{sum-integers (+ 1 a) b})))))

(\texttt{define (sum-squares a b)}
  (if (> a b)
   0
   (+ (square a)
       (\texttt{sum-squares (+ 1 a) b})))))

(\texttt{define (pi-sum a b)}
  (if (> a b)
   0
   (+ (/ 1 (square a))
       (\texttt{pi-sum (+ a 2) b})))))

Let's examine this new procedure

(\texttt{define (sum term a next b)}
  (if (> a b)
   0
   (+ (term a)
       (\texttt{sum term (next a) next b})))))

Higher order procedures

- A higher order procedure: takes a procedure as an argument or returns one as a value

(\texttt{define (sum-integers1 a b)}
  (sum \texttt{(lambda (x) x)} a \texttt{(lambda (x) (+ x 1))} b))

(\texttt{define (sum-squares1 a b)}
  (sum \texttt{square} a \texttt{(lambda (x) (+ x 1))} b))

(\texttt{define (pi-sum1 a b)}
  (sum \texttt{(lambda (x) (/ 1 (square x))}} a \texttt{(lambda (x) (+ x 2))} b))

Higher order procedures

- Takes a procedure as an argument or returns one as a value

(\texttt{define (sum-integers1 a b)}
  (sum \texttt{(lambda (x) x)} a \texttt{(lambda (x) (+ x 1))} b))

(\texttt{define (sum-squares1 a b)}
  (sum \texttt{square} a \texttt{(lambda (x) (+ x 1))} b))

(\texttt{define (add1 x) (+ x 1))

(\texttt{define (add2 x) (+ x 2))

(\texttt{define (pi-sum1 a b)}
  (sum \texttt{(lambda (x) (/ 1 (square x))}} a \texttt{add2} b))

Higher order procedures

- Takes a procedure as an argument or returns one as a value

(\texttt{define (sum-integers1 a b)}
  (sum \texttt{(lambda (x) x)} a \texttt{(lambda (x) (+ x 1))} b))

(\texttt{define (sum-squares1 a b)}
  (sum \texttt{square} a \texttt{(lambda (x) (+ x 1))} b))

(\texttt{define (add1 x) (+ x 1))

(\texttt{define (add2 x) (+ x 2))

(\texttt{define (pi-sum1 a b)}
  (sum \texttt{(lambda (x) (/ 1 (square x))}} a \texttt{add2} b))
Returning A Procedure As A Value

(define (add1 x) (+ x 1))
(define (add2 x) (+ x 2))
(define (addn x n) (+ x n))

(define (sum term a next b)
  (if (> a b)
    0
    (+ (term a)(sum term (next a) next b)))))

Returning A Procedure As A Value

(define incrementby
  (lambda(n)(lambda(x)(+x n))))

(define f1 (incrementby 6))
(define f2 (incrementby 2))

Computing derivatives

• A good approximation:
  \[ Df(x) \approx \frac{f(x + \epsilon) - f(x)}{\epsilon} \]

(define deriv
  (lambda (f)
    (lambda (x) (/ (- (f (+ x epsilon)) (f x)) epsilon))))

Using “deriv”

(define square (lambda (y) (* y y)))
(define epsilon 0.001)
(deriv square 5)
Common Pattern #1: Transforming a List

```scheme
(define (square-list lst)
  (if (null? lst)
      nil
      (adjoin (square (first lst))
               (square-list (rest lst))))
)

(define (double-list lst)
  (if (null? lst)
      nil
      (adjoin (* 2 (first lst))
               (double-list (rest lst))))
)

(define (MAP proc lst)
  (if (null? lst)
      nil
      (adjoin (proc (first lst))
               (map proc (rest lst))))
)

(define (square-list lst)
  (map square lst))

(define (double-list lst)
  (map (lambda (x) (* 2 x)) lst))
```

Transforms a list to a list, replacing each value by the procedure applied to that value.

Using common patterns over data structures

- We can more compactly capture our earlier ideas about common patterns using these general procedures.
- Suppose we want to compute a particular kind of summation:

\[
\sum_{i=0}^{n} f(a + i\delta) = f(a) + f(a + \delta) + f(a + 2\delta) + \ldots + f(a + n\delta)
\]

Using common patterns over data structures

```scheme
(define (generate-interval a b)
  (if (> a b)
      nil
      (cons a (generate-interval (+ 1 a) b))))

(define (integral f a b n)
  (let ((delta (/ (- b a) n)))
    (* delta (sum f a delta n))
  ))
```

Computing Integrals

```scheme
(define (integral f a b n)
  (let ((delta (/ (- b a) n)))
    (* delta (sum f a delta n)))
)
```

Integration as a procedure

Integration under a curve $f$ is given roughly by

\[
\int_{a}^{b} f(a) + f(a + dx) + f(a + 2dx) + \ldots + f(b)
\]
Procedures as arguments: a more complex example

- (define compose (lambda (f g x) (f (g x))))
- (compose square double 3)
- (square (* 3 2))
- (square 6)
- (* 6 6)
- 36

What is the type of compose? Is it:

(number → number), (number → number), number → number

Compose works on other types too

(define compose (lambda (f g x) (f (g x))))

(compose (lambda (p) (if p "hi" "bye"))
(lambda (x) (> x 0))
-5)  ==> "bye"

Will any call to compose work?

(compose < square 5)

(compose square double "hi")

Type of compose

(define compose (lambda (f g x) (f (g x))))

- Use type variables.
- compose:  (B → C), (A → B), A → C
- Meaning of type variables:
  All places where a given type variable appears must
  match when you fill in the actual operand types
- The constraints are:
  - F and G must be functions of one argument
  - the argument type of G matches the type of X
  - the argument type of F matches the result type of G
  - the result type of compose is the result type of F

Higher order procedures

- Procedures may be passed in as arguments
- Procedures may be returned as values
- Procedures may be used as parts of data structures
- Procedures are first class objects in Scheme!!