Procedures

CSE 413, Autumn 2005
Programming Languages

http://www.cs.washington.edu/education/courses/413/05au/

Combinations

- (operator operand operand)
- There are numerous pre-defined operators
- We can define our own, arbitrarily complex operators (functions, procedures) as well
- This is a key capability by which we can operate at higher levels of abstraction

Recall the `define` special form

- Special forms have unique evaluation rules
- `(define x 3)` is an example of a special form; it is not a combination
  - the evaluation rule for a simple define is "associate the given name with the given value"

References

- Sections 1.1.6-1.1.8, *Structure and Interpretation of Computer Programs*
- Section 4.1, *Revised Report on the Algorithmic Language Scheme (R5RS)*
Define and name a variable

- `(define <name> <expr>)`
  - `define` - special form
  - `name` - name that the value of `expr` is bound to
  - `expr` - expression that is evaluated to give the value for `name`

- `define` is valid only at the top level of a `<program>` and at the beginning of a `<body>`

Define and name a procedure

- `(define (<name> <formal params>) <body>)`
  - `define` - special form
  - `name` - the name that the procedure is bound to
  - `formal params` - names used within the body of procedure
  - `body` - expression (or sequence of expressions) that will be evaluated when the procedure is called.

  » The result of the last expression in the body will be returned as the result of the procedure call

Example definitions

```
(define pi 3.1415926535)
```

```
(define (area-of-disk r)
  (* pi (* r r)))
```

```
(define (area-of-ring outer inner)
  (- (area-of-disk outer)
     (area-of-disk inner)))
```

Defined procedures are "first class"

- Compound procedures that we define are used exactly the same way the primitive procedures provided in Scheme are used

  » names of built-in procedures are not treated specially; they are simply names that have been pre-defined

  » you can’t tell whether a name stands for a primitive (built-in) procedure or a compound (defined) procedure by looking at the name or how it is used
Evaluation example

- \((\text{area-of-ring } 4 \ 1)\)
  » evaluate operator \textit{area-of-ring} => procedure definition
  » evaluate 4 => 4
  » evaluate 1 => 1
  » apply the procedure to the arguments

Booleans

- Recall that one type of data object is boolean
  » \#t (true) or \#f (false)
- We can use these explicitly or by calculating them in expressions that yield boolean values
- An expression that yields a true or false value is called a predicate
  » \#t => \#t
  » \(< 5 \ 5) => \#f
  » \(> \pi \ 0) => \#t

Conditional expressions

- As in all languages, we need to be able to make decisions based on inputs and do something depending on the result
- A predicate expression is evaluated
  » true or false
- The consequent expression is evaluated if the predicate is true

Special form: \texttt{cond}

- \(\text{cond} \langle\text{clause}_1\rangle \langle\text{clause}_2\rangle \ldots \langle\text{clause}_n\rangle\)
- each clause is of the form
  » \(\langle\text{predicate}\rangle \langle\text{expression}\rangle\)
  » where \(\langle\text{predicate}\rangle\) is a boolean expression and \(\langle\text{expression}\rangle\) is the consequent expression to execute if \(\langle\text{predicate}\rangle\) is true
- the last clause can be of the form
  » \(\text{else} \langle\text{expression}\rangle\)
  » in which case \(\langle\text{expression}\rangle\) is executed if none of the preceding \(\langle\text{predicates}\rangle\) were true
**Example: sign.scm**

; return the sign of x as -1, 0, or 1

(define (sign x)
    (cond
        ((< x 0) -1)
        ((= x 0) 0)
        ((> x 0) +1))))

**Special form: if**

- (if <predicate> <consequent> <alternate>)
- (if <predicate> <consequent> )

- <predicate> is a boolean expression
- <consequent> is the expression to execute if <predicate> is true
- <alternate> is the expression to execute if <predicate> is false

**Examples: abs.scm, true-false.scm**

; absolute value function
(define (abs a)
    (if (< a 0)
        (- a)
        a))

; return 1 if arg is true, 0 if arg is false
(define (true-false arg)
    (if arg 1 0))

**Logical composition**

- (and <e₁> <e₂> ... <eₙ>)
- (or <e₁> <e₂> ... <eₙ>)
- (not <e>)

- Scheme interprets the expressions eᵢ one at a time in left-to-right order until it can tell the correct answer
  - ie, these are short-circuit operators
in-range.scm

; true if val is lo <= val <= hi

(define (in-range lo val hi)
  (and (<= lo val)
       (<= val hi)))

Newton's method for square root

• Guess a value y for the square root of x
• Is it close enough to the desired value \( \sqrt{x} \)?
  » ie, is \( y^2 \) close to \( x \)?
• If yes, then done. Return recent guess.
• If no, then new guess is average of current guess and \( \frac{x}{\text{guess}} \)
• Repeat with new guess

sqrta.scm

; Square root using Newton’s method

(define (average a b)
  (/ (+ a b) 2.0))

(define (good-enough? guess x)
  (< (abs (- (* guess guess) x)) 0.001))

(define (improve guess x)
  (average guess (/ x guess)))

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))

(define (sqrta x)
  (sqrt-iter 1.0 x))

auxiliary functions

; Square root using Newton’s method

(define (average a b)
  (/ (+ a b) 2.0))

(define (good-enough? guess x)
  (< (abs (- (* guess guess) x)) 0.001))

(define (improve guess x)
  (average guess (/ x guess)))
iterator and main functions

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x )))

(define (sqrt x)
  (sqrt-iter 1.0 x))

sqrt-iter

• Our first example of recursion
• Note that this recursion is used to implement a loop (an iteration)
  » We will see this over and over in Scheme
• Iteration is calling the same block of code with a changing set of parameters
• The syntax of the procedure is recursive but the resulting process is iterative
  » more on this next lecture