Procedures

CSE 413, Autumn 2002
Programming Languages

http://www.cs.washington.edu/education/courses/413/02au/

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

Readings and References

• Reading
  » Sections 1.1.6-1.1.8, Structure and Interpretation of
    Computer Programs, by Abelson, Sussman, and Sussman

• Other References
  » Section 4.1, Revised\textsuperscript{5} Report on the Algorithmic Language
    Scheme (R5RS)

Define and name a procedure

• \texttt{(define (name formal params) body)}
  
  » \texttt{define} - special form
  
  » \texttt{name} - the name that the procedure is bound to
  
  » \texttt{formal params} - names used within the body of
    procedure
  
  » \texttt{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

```
(area-of-ring 4 1)
```

- evaluate operator `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: cond

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

- \( \text{if} \ (\text{predicate} \ \text{consequent} \ \text{alternate}) \)
- \( \text{if} \ (\text{predicate} \ \text{consequent}) \ )
- \( \text{predicate} \) is a boolean expression
- \( \text{consequent} \) is the expression to execute if \( \text{predicate} \) is true
- \( \text{alternate} \) is the expression to execute if \( \text{predicate} \) 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 \langle e_1 \rangle \langle e_2 \rangle ... \langle e_n \rangle)
- (or \langle e_1 \rangle \langle e_2 \rangle ... \langle e_n \rangle)
- (not \langle e \rangle)

- Scheme interprets the expressions $e_i$ one at a time in left-to-right order until it can tell the correct answer
  » i.e., 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 $\sqrt{\frac{2}{x}}$ ?
  » i.e., is $y^2$ close to $x$?
- If yes, then done. Return recent guess.
- If no, then new guess is average of current $\frac{x}{\text{guess}}$ and $\text{guess}$
- 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 (sqrta 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