More Procedures and lists

- Iterative vs Recursive Procedures
- Syntactic sugar and shortcuts
- Lists
Recursive Procedures

• Let’s look at this one again:
• (define (sumto n)
  (if (= n 1)
    1
    (+ n (sumto (- n 1))))


Recursive Procedures

• What happens when n is very large?
  (define (sumto n)
    (if (= n 1)
      1
      (+ n (sumto (- n 1))))))

• (+ 100 (+ 99 (+ 98 (+ 97 ....

Recursive Process

- \((+ 100 (+ 99 (+ 98 (+ 97 (sumto 96)))))\)
- Each addition is a pending operation
- Interpreter has to store it, wait to evaluate it
- Try it for a really large \(n\) -- should slow down fast
Newer Version

- Old

\[
\text{(define (sumto n)}
\]

\[
\text{(if (= n 1) 1)}
\]

\[
\text{(+ n (sumto (- n 1))}]
\]

- New

\[
\text{(define (sumto n ans)}
\]

\[
\text{(if (= n 1) (+ ans 1)}
\]

\[
\text{(sumto (- n 1) (+ ans n))}]
\]

Are there any more pending operations?
Iterative vs Recursive

- Two types of processes:
  - Iterative: no pending operations
  - Recursive: pending operations
- Hint: look for the point where the procedure is called again:
  - `(op (call-again ....))` -- recursive
  - `(call-again …)` -- iterative
Friendlier Iterative Procedures

- Extra arguments are annoying
- Use a helper function instead
- `(define (sum-to-helper n ans)
  (if (= n 1)
    (+ ans 1)
    (sum-to-helper (- n 1) (+ ans n))))`
- `(define (sum-to n)
  (sum-to-helper n 0))`
Iterative versions

• Factorial:
  – (define (fact-iter n) …

• Exponentiation:
  – (define (expt-iter x n) …
Syntactic Sugar

• Syntax:
  – How to correctly arrange the language to describe computation

• Syntactic Sugar “sweetens” language to make it more convenient
  – No new capabilities
First example: \texttt{let}

- \texttt{(let ((name1 \texttt{val1})
               (name2 \texttt{val2}))
   expr)}
- Binds the value of \texttt{val1} to \texttt{name1} when evaluating \texttt{expr}
  - No changes elsewhere
Let example

• (let ((a 3)
        (b 5))
  (+ a b))
Let example

• (let ((a 3)
  (b 5))
  (+ a b))

• Equivalent to:

• ((lambda (a b) (+ a b))
  3 5)
Let “practice”

- (let (((+ *)
          (* +))
       (+ 3 (* 4 5)))))
Let practice

- (define m 3)
  (let ((m (+ m 1)))
    (+ m 1))
(define n 4)
(let ((n 12)
  (o (+ n 2)))
(* n o))
More Syntactic Sugar

• This one appears quite often:
  (define new-function
    (lambda (a b c) exprs)

• Shortcut version:
  (define (new-fuction a b c) exprs)
Lists

• Basic data structure in Scheme
• Create a list using list
  – (list 1 2 3 4 5 6)
• (define lst (list 1 2 3 4 5))
How are lists represented

• (list 1 2 3 4)

Null, or “empty list”
Lists

- Anything can be in a list
- \( (\text{define } \text{lst1} \ (\text{list} \ (\text{list} \ 1 \ 2) \ (\text{list} \ 3 \ 4))) \)
Pairs

- A list is built out of pairs
- A pair is a basic type
- Box with two pointers
- To create a pair use `cons`
- Also known as a “cons cell”
• `(cons 1 2)`

• `(cons 1 null)`
Building lists

• We can use **cons** to build up lists
• Draw a box and pointer diagram for
• (cons 1 (cons 2 (cons 3 (cons 4 null)))))
Getting stuff out of pairs

• We can put things in a cons object, but how do we get them back?

car  

\[
\text{cons object}
\]

cdr
Examples

(car (cons 1 2))
(cdr (cons 1 2))

(define lst (list 1 2))
(car lst)
(cdr lst)
(cdr (cdr lst))
Finding the length of a list

- Define a procedure length
- What’s the recursive case?
- What’s the base case?
Checking for the empty list

- Base case is an empty list
- Check for it using the `null?` Predicate

- `(null? lst)` returns `#t` if `lst` is the empty list, and `#f` otherwise
length

• Plan
  – Base Case: Empty list \( \rightarrow 0 \)
  – Recursive Case: \( 1 + \text{length of “rest of list”} \)
length

• (define (length list)