Topic #4:
Pairs & Lists

CSE 413, Autumn 2004

Programming Languages

http://www.cs.washington.edu/education/courses/413/04au/

References

- Section 15.5, *Concepts of Programming Languages*
- Sections 6.3.2, *Revised^5 Report on the Algorithmic Language Scheme (R5RS)*
- For more:
  - » Sections 2-2.2.1, *Structure and Interpretation of Computer Programs*

Procedural abstractions

- So far, we have talked about primitive data elements and done various levels of abstraction using procedures only
  - » This is a key capability in being able to recognize and implement common behaviors
- The ability to combine data elements will further extend our ability to model the world

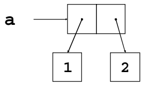
Compound data

- In order to build compound structures we need a way to combine elements and refer to them as a single blob
- We can write a `lambda` expression that combines one or more expressions
- We can write a `cons` expression that ties two data elements together

`(cons a b)`

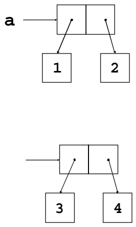
- Takes `a` and `b` as args, returns a compound data object that contains `a` and `b` as its parts
- We can extract the two parts with accessor functions `car` and `cdr` ("could-er")

```
(define a (cons 1 2))
```



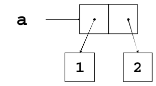
`car` and `cdr`

```
(define a (cons 1 2))

(car a)

(cdr a)


(car (cons 3 4))

(cdr (cons 3 4))
```

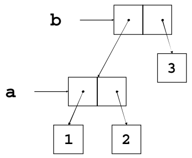


`car` and `cdr`

```
(define a (cons 1 2))
```



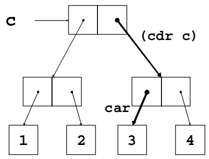
```
(define b (cons a 3))

(car (car b))
(cdr (car b))
(cdr b)
```



`(car (cdr c))`

```
(define c (cons (cons 1 2) (cons 3 4)))
```

```
(car (car c))
(cdr (car c))
(car (cdr c))
(cdr (cdr c))
```

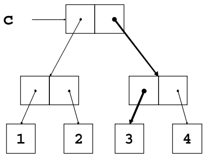


`(cadr c)`

- We can abbreviate the repeated use of car and cdr

```
(define c (cons (cons 1 2) (cons 3 4)))
```

```
(caar c)
(cdar c)
(cadr c)
(cddr c)
```

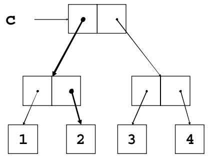


pair? predicate

- (pair? z) is true if z is a pair

```
(define c (cons (cons 1 2) (cons 3 4)))
```

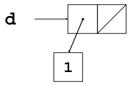
```
(pair? c)
(pair? (car c))
(pair? (cdr c))
(pair? (caar c))
(pair? (cdar c))
```



nil

- if there is no element present for the car or cdr branch of a pair, we indicate that with the value nil
  - nil (or null) represents the empty list '()
- (null? z) is true if z is nil

```
(define d (cons 1 '()))
(car d)
(cdr d)
(null? (car d))
(null? (cdr d))
```



What do we really know about pairs?

- An Application Programming Interface (API)
  - `cons` - constructor
  - `car`, `cdr` - accessor functions
- We may think we know how they are stored
  - box-and-pointer drawings
  - pointers to pointer blocks ...
- But if we can stay at the API level, the separation between layers of implementation can stay clean which is a "good thing"

Can we implement cons/car/cdr?

- If we focus on the behaviors that are defined what do we actually need to do?
- **`(cons a b)`**
- **`(car`** ***something*****`)`**
- **`(cdr`** ***something*****`)`**

---

something

- We tend to think of the *something* returned by `cons` as a structured data variable of some sort
- However, the only actual requirement on *something* is that we can recover `a` and `b` from it using procedures named `car` and `cdr`
- How about we use a procedure definition for *something* ...

Procedural representation of pairs

definition

```
(define (cons x y)
  (lambda (m) (m x y)))

(define (car z)
  (z (lambda (p q) p)))

(define (cdr z)
  (z (lambda (p q) q)))
```

usage

```
(define a (cons 1 2))
(car a)
(cdr a)
```

Procedural `cons` and `car`

`cons`

```
(define (cons x y)
  (lambda (m) (m x y)))
```

`car`

```
(define (car z)
  (z (lambda (p q) p)))
```

Lexical closure

- Take another look at the definition of cons

```
(define (cons x y)
  (lambda (m) (m x y)))
(define (car z)
  (z (lambda (p q) p)))
```

- Where did the values of x and y come from?
- Are they still around when we call `car` / `cdr`?

---

current symbol definitions

- Lambda expressions evaluate to what is called a lexical closure
  - » a coupling of code and a lexical environment (a scope)
  - » The lexical environment is necessary because the code needs a place to look up the definitions of symbols it references

definition and execution

```
(define (cons x y)
  (lambda (m) (m x y)))
```

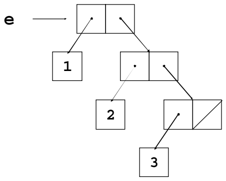
- x and y are referenced in the environment of the lambda expression's definition
  - » its lexical environment, which is in the definition of `cons`
- not the environment of its execution
  - » its dynamic environment, which is in `car`

Pairs are the glue

- Using `cons` to build pairs, we can build data structures of unlimited complexity
- We can roll our own
- We can adopt a standard and use it for the basic elements of more complex structures

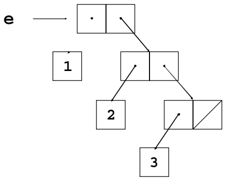
Lists

- By convention, a list is a sequence of linked pairs
  - » `car` of each pair is the data element
  - » `cdr` of each pair points to list tail or the empty list



List construction

```
(define e (cons 1 (cons 2 (cons 3 '()))))
```



```
(define e (list 1 2 3))
```

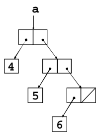
procedure `list`

(list *a b c* ...)

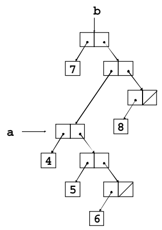
- `list` returns a newly allocated list of its arguments
  - » the arguments can be atomic items like numbers or quoted symbols
  - » the arguments can be other lists
- The backbone structure of a list is always the same
  - » a sequence of linked pairs, ending with a pointer to null (the empty list)
  - » the `car` element of each pair is the list item
  - » the list items can be other lists

List structure

```
(define a (list 4 5 6))
```

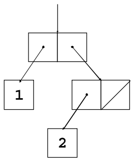


```
(define b (list 7 a 8))
```



Examples of list building

```
(cons 1 (cons 2 '()))

(cons 1 (list 2))

(list 1 2)
```



How to process lists?

- A list is zero or more connected pairs
- Each node is a pair
- Thus the parts of a list (this pair, following pairs) are lists
- A natural way to express list operations?

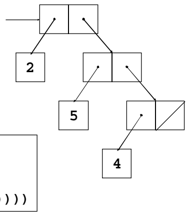
`cdr` down

```
(define (length m)
  (if (null? m)
      0
      (+ 1 (length (cdr m)))))
```

---

sum the items in a list

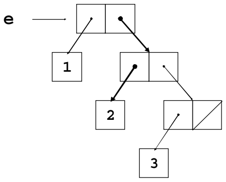
```
(add-items (list 2 5 4))
```



```
(define (add-items m)
  (if (null? m)
      0
      (+ (car m) (add-items (cdr m)))))
```

Yet another list function

```
(define e (cons 1 (cons 2 (cons 3 '()))))

  (define (zip z)
    (if (pair? z)
        (begin
          (display (car z))
          (display " ")
          (zip (cdr z)))
        (newline)))

  (zip e)
```



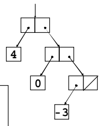
`cons` up

- We can build a list to return to the caller piece by piece as we go along through the input list

```
(define (reverse m)
  (define (iter shrnk grow)
    (if (null? shrnk)
        grow
        (iter (cdr shrnk) (cons (car shrnk) grow))))
  (iter m '()))
```

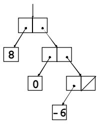
multiply each list element by 2

```
(double-all (list 4 0 -3))
```



```
(define (double-all m)
  (if (null? m)
      '()
      (cons (* 2 (car m)) (double-all (cdr m)))))
```

```
(cons 8 (cons 0 (cons -6 '())))
```



Variable number of arguments

- We can define a procedure that has zero or more required parameters, plus provision for a variable number of parameters to follow
  - » The required parameters are named in the `define` statement as usual
  - » They are followed by a "." and a single parameter name
- At runtime, the single parameter name will be given a list of all the remaining actual parameter values

`(same-parity x . y)`

```
(define (same-parity x . y)
…


> (same-parity 1 2 3 4 5 6 7)
(1 3 5 7)
> (same-parity 2 3 4 5 6 7)
(2 4 6)
>
```

The first argument value is assigned to x, all the rest are assigned as a list to y

`map`

- We can use the general purpose function `map` to map over the elements of a list and apply some function to them

```
(define (map p m)
  (if (null? m)
      '()
      (cons (p (car m))
            (map p (cdr m)))))


(define (double-all m)
  (map (lambda (x) (* 2 x)) m))
```