Pairs

CSE 413, Autumn 2002
Programming Languages

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

Readings and References

- Reading
  - » Sections 2-2.1.3, *Structure and Interpretation of Computer Programs*, by Abelson, Sussman, and Sussman
- Other References
  - » Section 6.3.2, *Revised^5 Report on the Algorithmic Language Scheme (R5RS)*

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

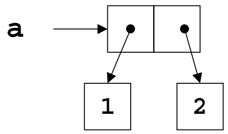
Compound data

- 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
  - » the resulting combination is a *procedure*
- We can write a `cons` expression that ties two data elements together
  - » the resulting combination is a *pair*

(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")

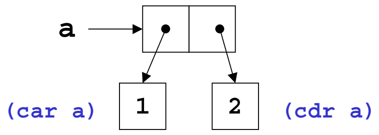
```
(define a (cons 1 2))
```



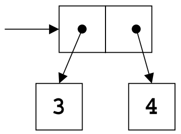
---

car and cdr

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

(car a)

(cdr a)
```



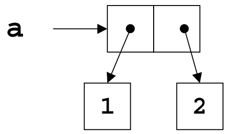
```
(car (cons 3 4))

(cdr (cons 3 4))
```



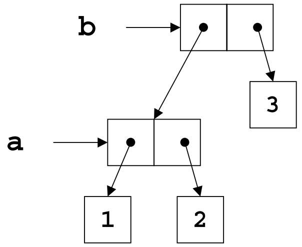
---

car and cdr

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



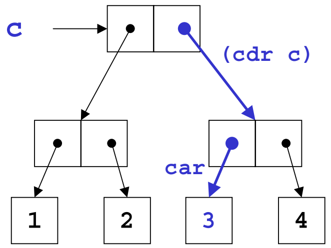
```
(define b (cons a 3))

(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))
```

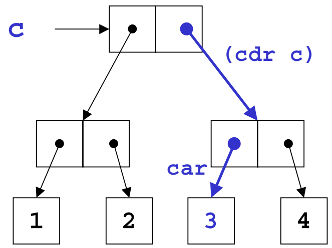


(cadr c)

- 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)
```



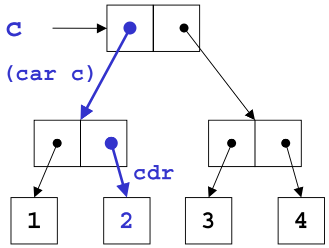
---

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))
```

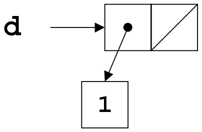


---

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))
```

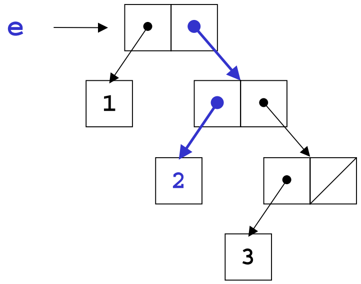


(cons 1 (cons 2 (cons 3 '())))

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

```
(car e)
(car (cdr e))
(car (cddr e))
```

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



Email from Steve Russell

- I wrote the first implementation of a LISP interpreter on the IBM 704 at MIT in early in 1959. I hand-compiled John McCarthy's "Universal LISP Function".
- The 704 family (704, 709, 7090) had "Address" and "Decrement" fields that were 15 bits long in some of the looping instructions. There were also special load and store instructions that moved these 15-bit addresses between memory and the index registers (3 on the 704, 7 on the others)
- We had devised a representation for list structure that took advantage of these instructions.
- Because of an unfortunate temporary lapse of inspiration, we couldn't think of any other names for the 2 pointers in a list node than "address" and "decrement", so we called the functions CAR for "Contents of Address of Register" and CDR for "Contents of Decrement of Register".
- After several months and giving a few classes in LISP, we realized that "first" and "rest" were better names, and we (John McCarthy, I and some of the rest of the AI Project) tried to get people to use them instead.
- Alas, it was too late! We couldn't make it stick at all. So we have CAR and CDR.

http://home.planet.nl/~faase009/HaCAR_CDR.html

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"

"Need to know" only

- As much as possible, the API should only expose the functions that the user needs in order to accomplish tasks with the logical data object we are defining
  - » If the implementation does not expose unnecessary details, then the user won't use them and you won't be stuck with them forever
- You want to be able to jack up the house and roll in a completely different foundation
  - » think about the evolution of device drivers
  - » open, read, write, status, control, close

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)`**
  - » define *something* that can be used later to extract `a` and `b`
- **`(car`** ***`something`*****`)`**
  - » recover `a` from *something*
- **`(cdr`** ***`something`*****`)`**
  - » recover `b` from *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

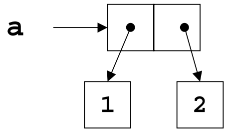
```
(define (cons x y)
  (lambda (m) (m x y)))

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

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

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

`(cons a b)`

- Takes `a` and `b` as args, returns a compound data object (aka *something*) 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 a)
(cdr a)
```



Procedural `cons` and `car`

`cons`

- » a procedure that takes two arguments x and y and returns a procedure
- » the returned procedure takes one argument (a function) and applies that function to the values of x and y

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

`car`

- » a procedure that takes one argument (the function defined by cons) and applies that function to a function that takes two arguments and returns the first one

```
(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?
  - » the initial call to `cons`, the definition call
- Are they still around when we call `car` and `cdr`?
  - » yes, they are part of the environment that is stored by the lambda definition statement in `cons`

---

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`