Principles of Programming Languages

Topic: Functional Programming A

Professor Lou Steinberg

Spring 2016

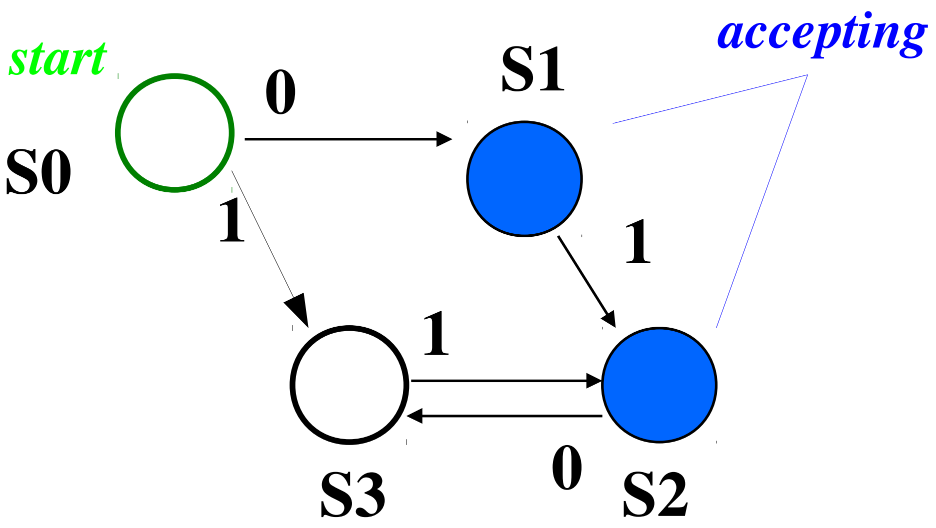
Review - Formal Languages

- A formal language is a set of strings
- To specify the language you must
  - specify the set of allowed characters,
  - give some way to tell if a string of these characters is in the language or not
- We have seen several ways to say whether a string is in a language
  - Grammar
  - Regular Expression
  - Automata

Review - Finite State Automata

- **Yet another way to specify what strings are in a language**
  - **also can specify exactly the same languages as regular grammars**

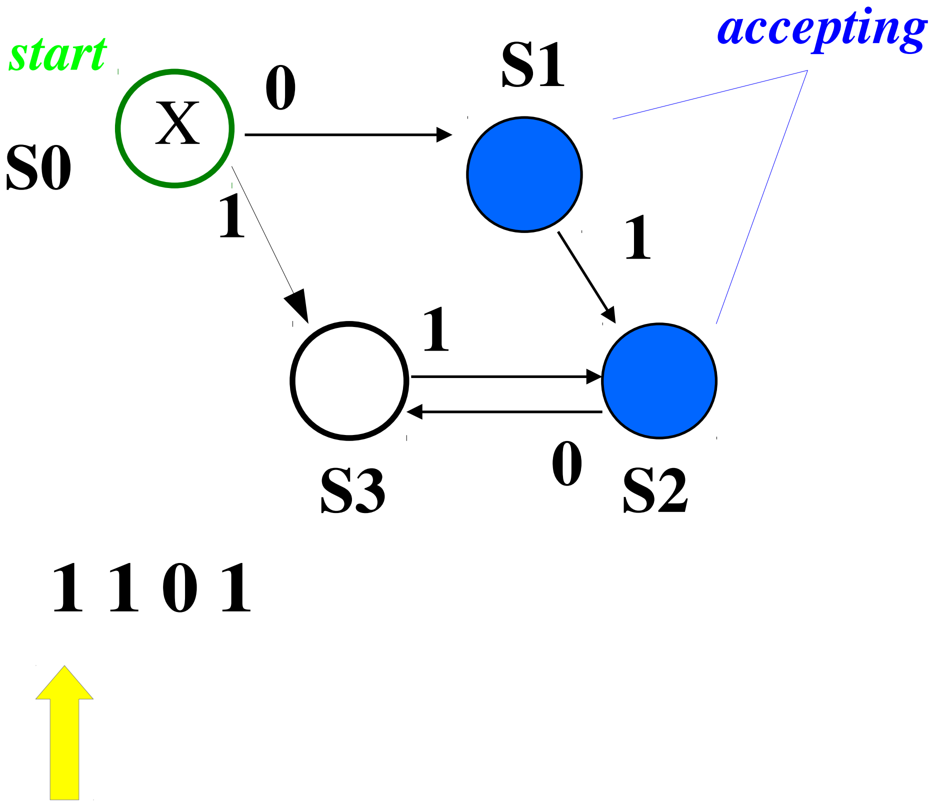
Finite State Automaton (FA)



transition table

states:	inputs: 0	1
S0	**S1**	**S3**
S1	**---**	**S2**
S2	**S3**	**---**
S3	**---**	**S2**

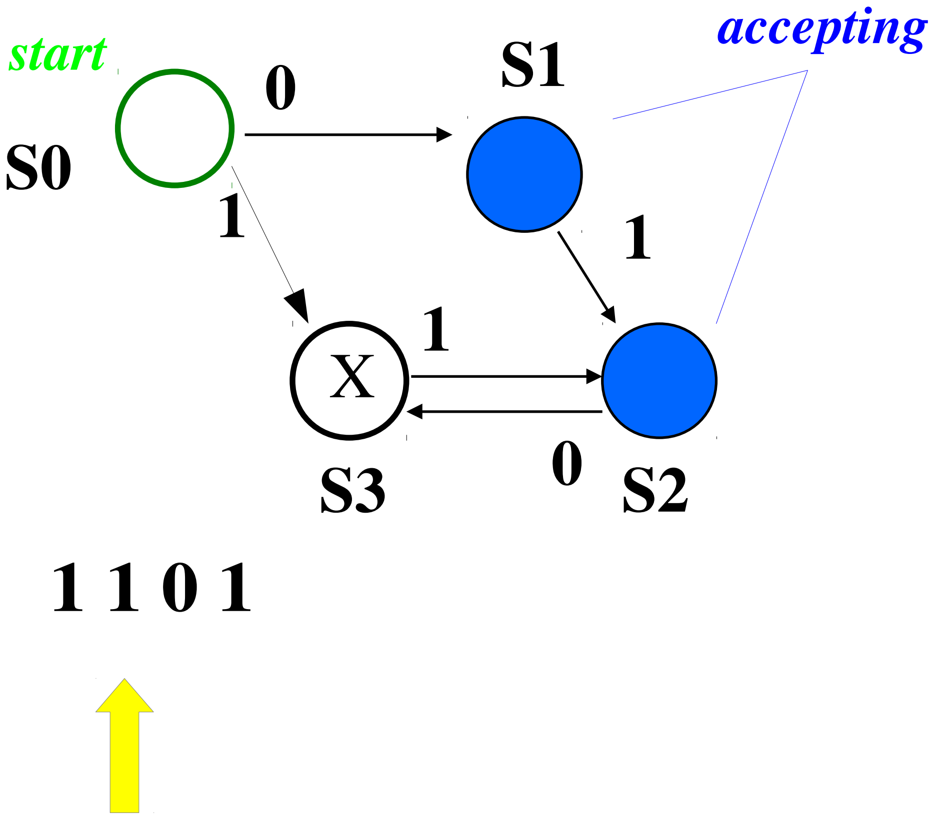
Finite State Automaton (FA)



transition table

states:	inputs: 0	1
S0	**S1**	**S3**
S1	**---**	**S2**
S2	**S3**	**---**
S3	**---**	**S2**

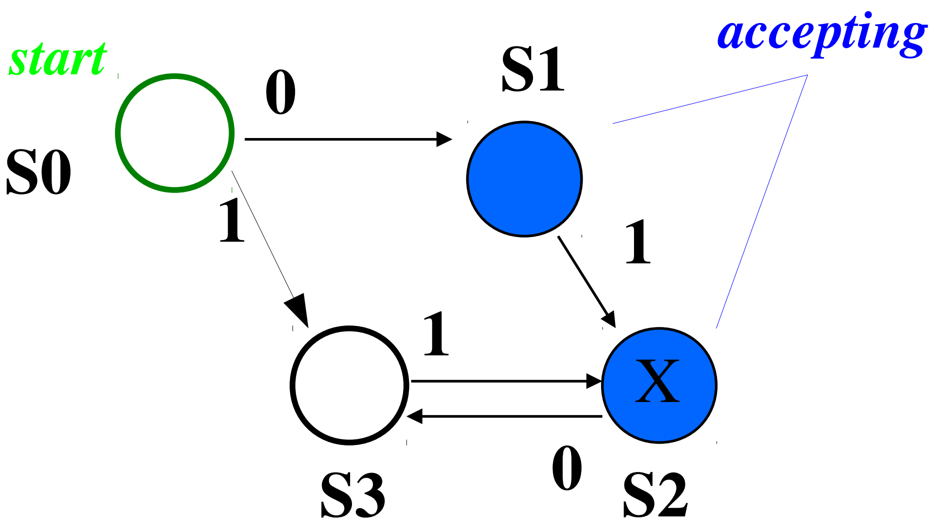
Finite State Automaton (FA)



transition table

states:	inputs: 0	1
S0	S1	S3
S1	---	S2
S2	S3	---
S3	---	S2

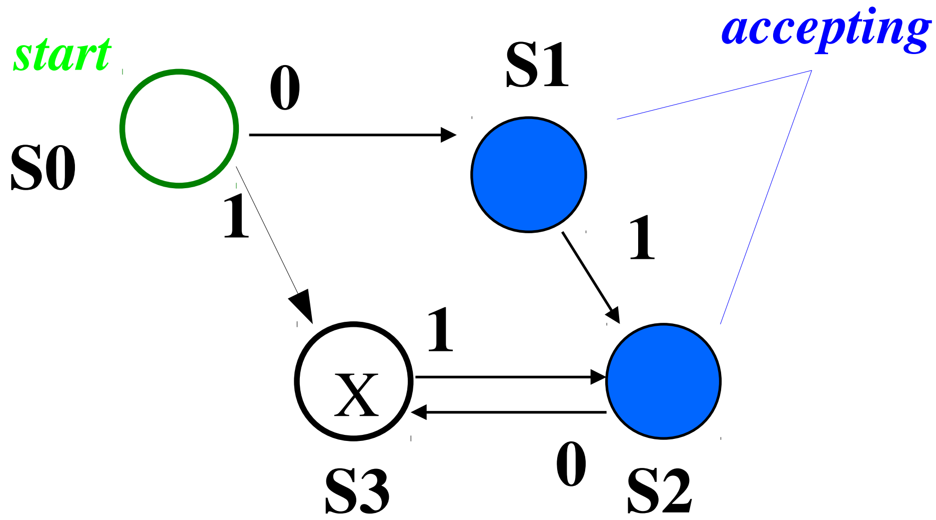
Finite State Automaton (FA)



1 1 0 1

	transition table	
	inputs:	
states:	0	1
S0	S1	S3
S1	---	S2
S2	S3	---
S3	---	S2

Finite State Automaton (FA)

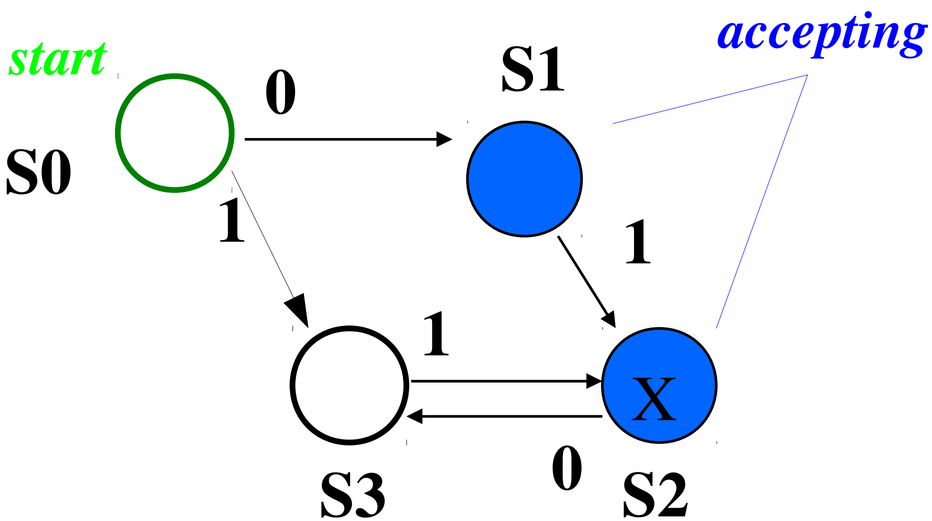


transition table

states:	inputs: 0	1
S0	**S1**	**S3**
S1	**---**	**S2**
S2	**S3**	**---**
S3	**---**	**S2**

1 1 0 1

Finite State Automaton (FA)

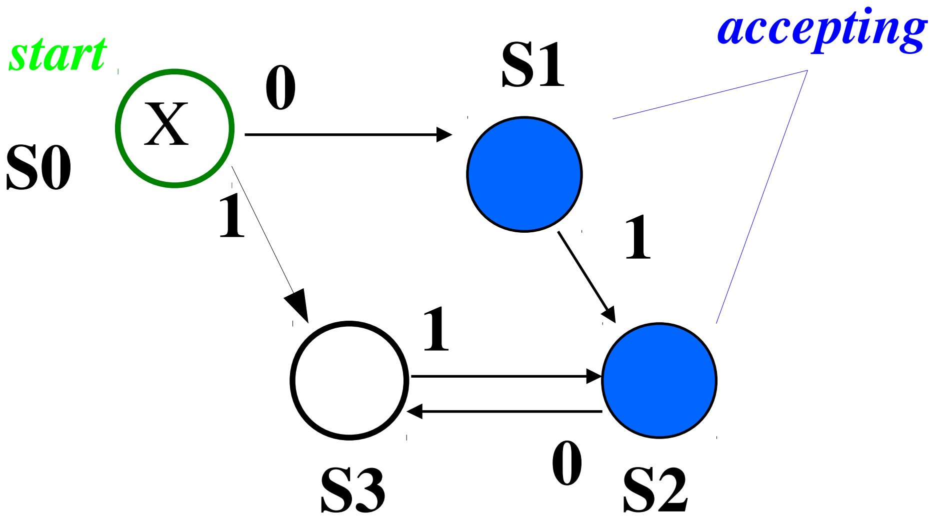


transition table

states:	inputs: 0	1
S0	**S1**	**S3**
S1	**---**	**S2**
S2	**S3**	**---**
S3	**---**	**S2**

1 1 0 1

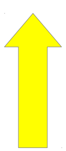
Finite State Automaton (FA)



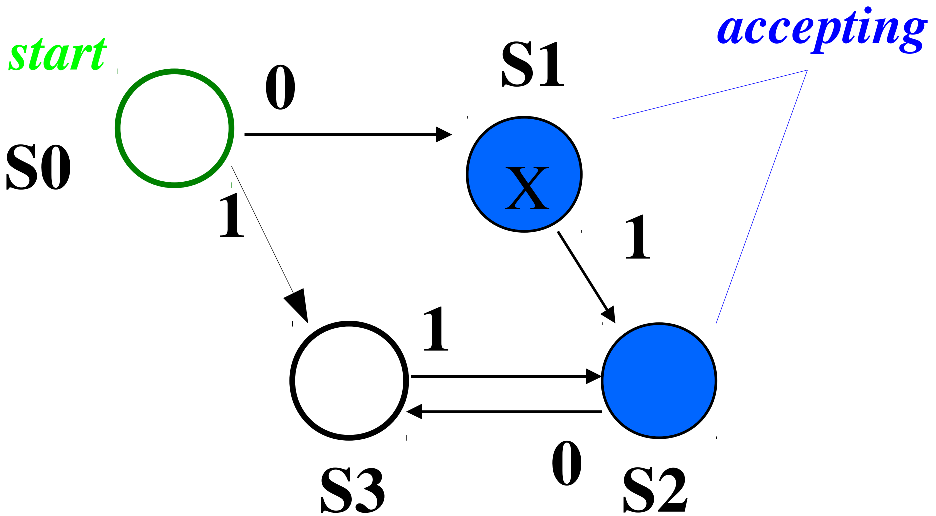
transition table

states:	inputs: 0	1
S0	**S1**	**S3**
S1	**---**	**S2**
S2	**S3**	**---**
S3	**---**	**S2**

0 1 1



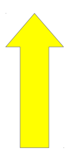
Finite State Automaton (FA)



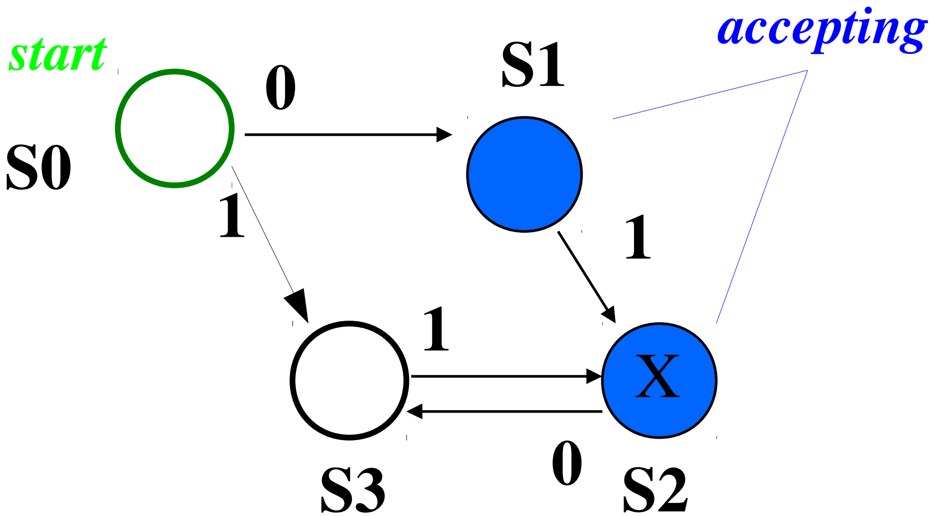
transition table

states:	inputs: 0	1
S0	**S1**	**S3**
S1	**---**	**S2**
S2	**S3**	**---**
S3	**---**	**S2**

0 1 1



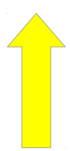
Finite State Automaton (FA)



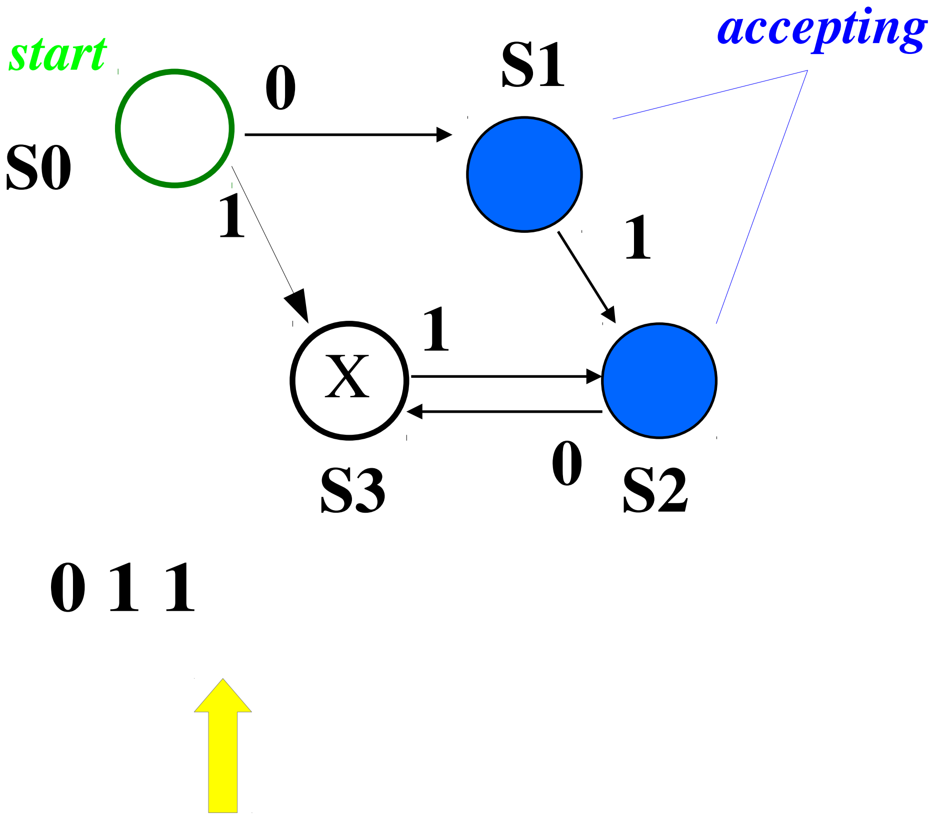
transition table

states:	inputs: 0	1
S0	**S1**	**S3**
S1	**---**	**S2**
S2	**S3**	**---**
S3	**---**	**S2**

0 1 1



Finite State Automaton (FA)

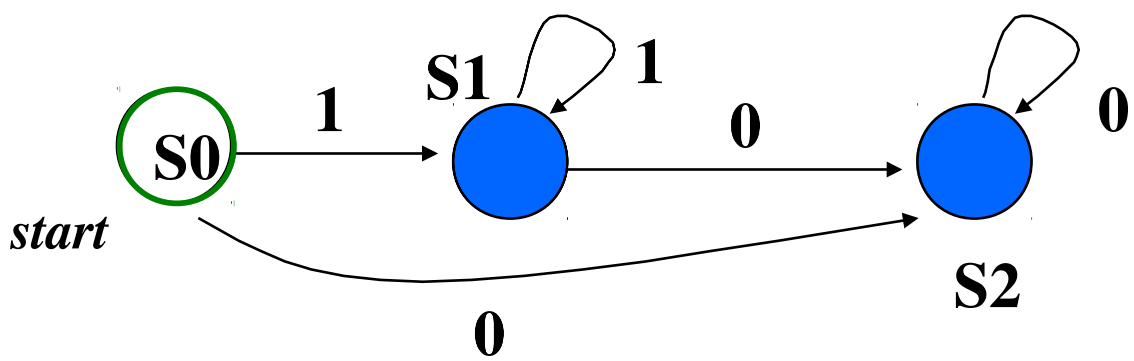


transition table

states:	inputs: 0	1
S0	**S1**	**S3**
S1	**---**	**S2**
S2	**S3**	**---**
S3	**---**	**S2**

Finite State Automaton (FA)

Binary numbers containing at least one digit, in which all the 1's precede all the 0's:



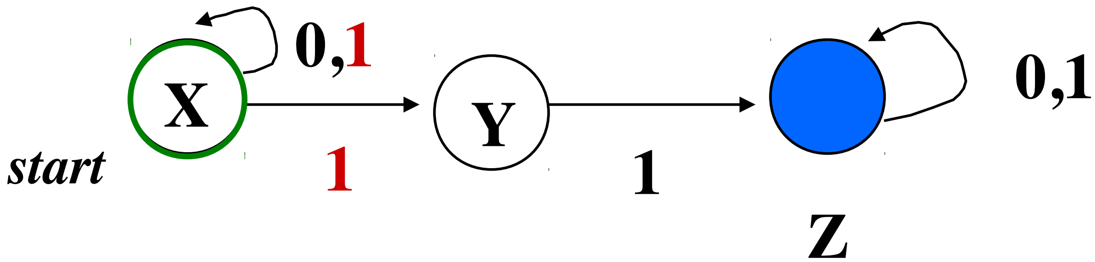
Recognizes: $(\; 0^{+} \;) \mid (\; 1^{+} \, 0^{*} \;)$

Review - Deterministic and Nondeterministic FAs

- **Deterministic FA: given state and character: one next state**
- **Nondeterministic FA: given state and character: may be more than one next state possible, and/or ε transitions**
  - **the FSA can choose which of the possible states to actually go to**
  - **if any sequence of choices results in an accepting state, string is in the language**

NFA

- **NFA:**



- **RE** $(0 \mid 1)^* \; 1 \; 1 \; (0 \mid 1)^*$

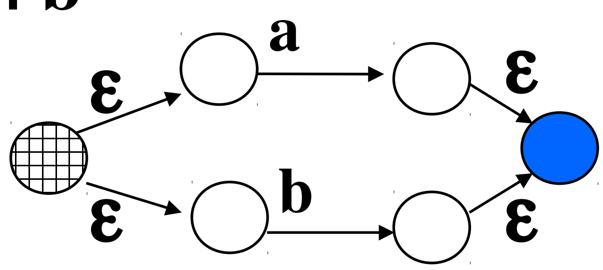
RE => NFA => DFA (=> RE)

- **If there is a RE that recognizes L,**
  Then there is a NFA that recognizes L
- **If there is an NFA that recognizes L,**
  Then there is a DFA that recognizes L
- **If there is a DFA that recognizes L,**
  Then there is a RE that recognizes L

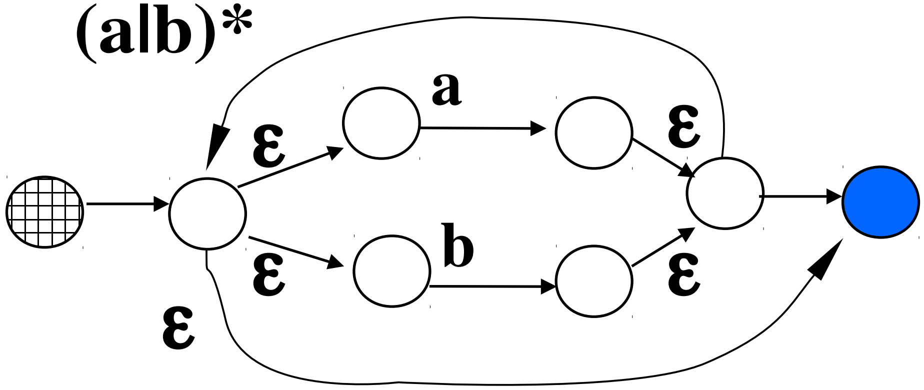
RE → NFA Example

- **Build NFA for RE (a | b)* a b (a | b)***

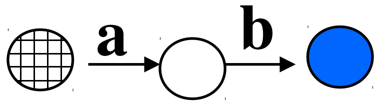
a | b



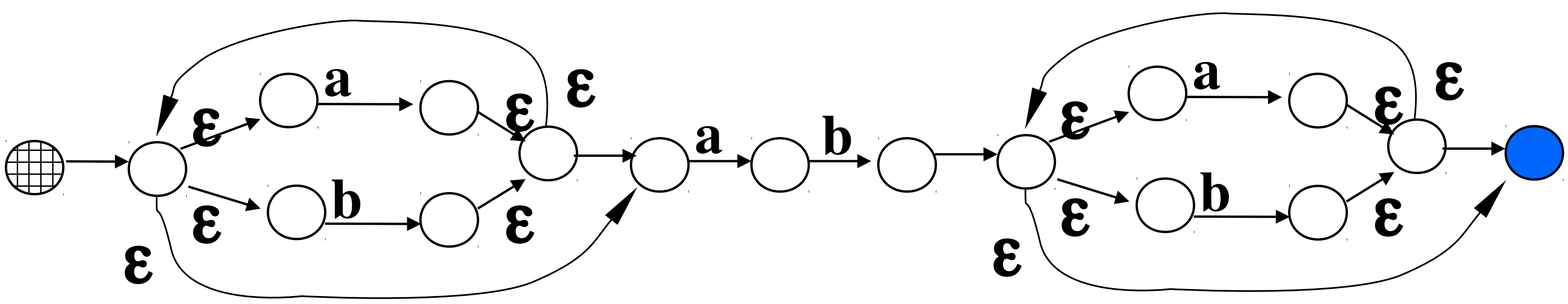
(a|b)*



a b

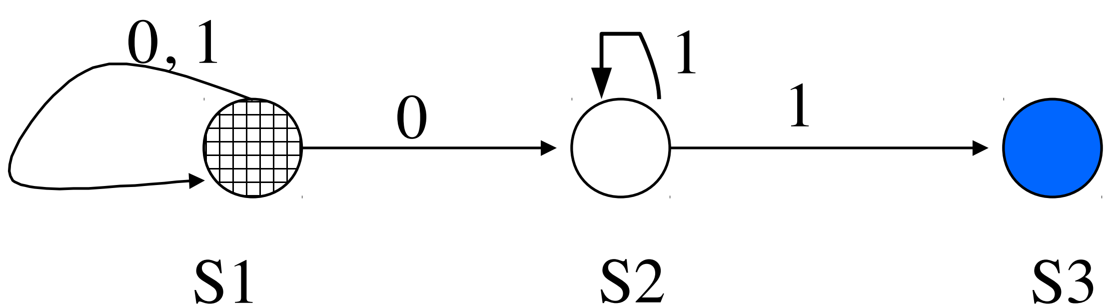


(a|b)* a b (a|b)*



NFA to DFA

- **Key idea: Each state in DFA corresponds to a *set* of states in the NFA**
- **If you are in a given state of the NFA it means you would be in one of the corresponding states of the DFA, depending on non-deterministic choices**



0 1 1

Not all languages have an FA

- **Palindrome: a string that reads the same backwards as forwards**
  - **0110**
  - **0010100**
- **There is no FA that accepts a string if and only if it is a palindrome**

No FA for palindromes

- **Basic idea:**
  - **The only memory a FA has of what it has already seen in the string is what state it is in**
  - **For any specific FA, the number of states is fixed**
  - **So, for any FA, a long enough string will make it run out of states to record the string-so-far**

No RE for palindromes

- **There is no RE that describes the language "strings that are palindromes"**
  - **Proof: If there was such an RE it could be translated into a FA that accepts a string if and only if it is a palindrome, and we just proved there was no such FA**

Tasks for REs and FAs

Things you need to know how to do for exams:

- **Recognition of a string**
  - **Is this given string in the language described (recognized) by this given RE (FA)?**
- **Description of a language**
  - **Given an RE (FA), what language does it describe (recognize)?**
- **Codification of a language**
  - **Given a language, find an RE and an FA that corresponds to it**

Scheme

- **To download Scheme see Sakai: Resources > Scheme > Scheme-links.html**
- **Scheme-links.html also has a link to *Sketchy Lisp*, a good, short, free book on Scheme**

Scheme

Scheme is a *dynamic* language

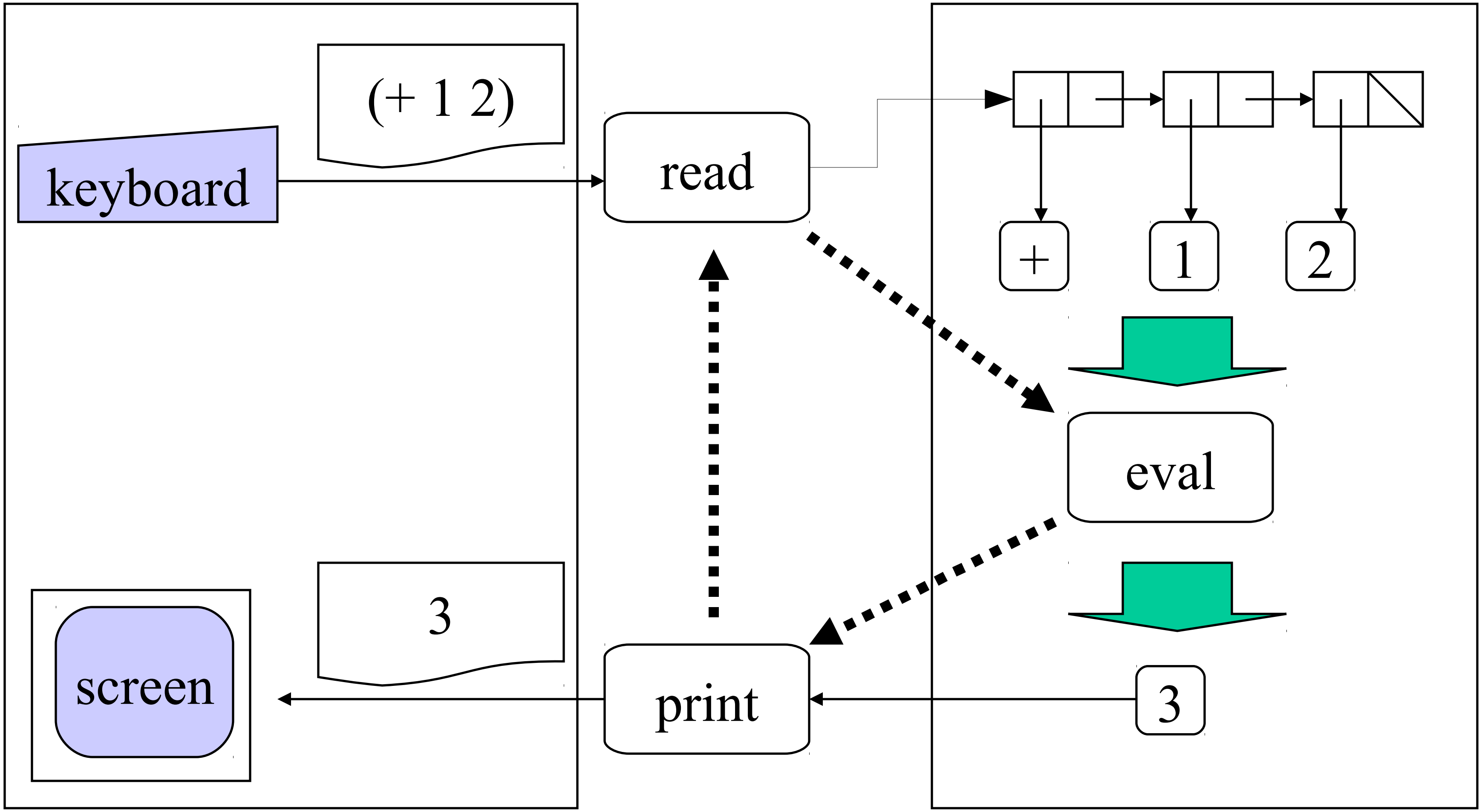
- **Blurs run time and compile time**

Scheme is an *interactive* language

- **Repeatedly**
  - **It reads an expression into internal form**
  - **It evaluates that expression**
  - **It prints the result of that evaluation**
- **“Read-eval-print loop”, or REPL**

Scheme is a *functional programming* language

Read-Eval-Print Loop



Scheme

A program is an expression to be evaluated, not a command to be executed.

An expression is:

- **A literal constant: 5, 3.1416, “hello”**
- **a variable that has been bound to some value: x, ?a, +**
- **or a function application**

A function application is written as a list: (+ 3 5)

- **parens around the list; white space separates elements**
- **the first element in the list specifies a function: +**
- **the remaining elements specify the arguments: 3, 5**

Function Application

To evaluate a function application, e.g., (+ 3 5):

- evaluate the first element: + is a predefined variable

  + => [machine code for addition]
- evaluate the remaining elements: 3 and 5 are literals

  3 => 3

  5 => 5
- apply the value of the first to the values of the rest

  apply [machine code for addition] to 3, 5 => 8

This is like the mathematical concept of a function:

sum(3, 5) = 8

Function Application

Function applications can be nested, e.g., (+ 5 (- 7 3)):

+ => [machine code for addition]

5 => 5

(- 7 3) => evaluate like any function application

- => [machine code for subtraction]

7 => 7

3 => 3

apply [machine code for subtraction] to 7 and 3

=> 4

apply [machine code for addition] to 5 and 4

=> 9

Special Forms

- **Exceptions to function application rules**
- **Special symbols as car of list**
  - **quote, if, lambda, a few others**
  - **Evaluation rules depend on car of list**
  - **quote: don't evaluate, just return arg**

    (quote a) => a

    (quote (+ 3 4)) => (+ 3 4) (a list, not a number)
  - **quote is extra special:**

    'a is read in as if it were (quote a)

    '(+ 3 4) is read in as if it were (quote (+ 3 4))

Function Abstraction

- **How do we get new functions? We define them by a process called function abstraction.**
  - **First, write an expression: (+ x x)**
  - **Then wrap it in a lambda form: (lambda (x) (+ x x))**
  - **This value of this lambda expression is a function with a formal parameter, x , and a body, (+ x x)**
  - **You can use this expression-whose-value-is-a function just like you can use a variable-whose-value-is-a-function:**

    ((lambda (x) (+ x x)) 4)

function

argument

Function Abstraction

How to evaluate `((lambda (x) (+ x x)) 4)` ?

- `(lambda (x) (+ x x))` ⇒ [function ...]
- `4` ⇒ `4`
- apply [function ...] to `4`
  - create a new binding context in which `x` has the value `4`,
    evaluate `(+ x x)` in this context:
    - `+` ⇒ [machine code for addition]
    - `x` ⇒ `4`
    - `x` ⇒ `4`
    - apply [machine code for addition] to `4` and `4` ⇒ `8`
  - ⇒ `8`

Function Definitions

- Note that

  (lambda ...)

  is a special form but

  ((lambda ...)   ...)

  is evaluated normally

Top-Level Definitions

- **(define v (+ 3 5)) assigns 8 to variable v**
- **(define double (lambda (x) (+ x x))) is similar**
- **(define (double x)(+ x x)))is shorthand for the line above**

Booleans and if

- **#f represents false, #t represents true**
  - **in fact, everything besides #f represents true**
- **(if (even? n) (/ n 2) (/ (- n 1) 2))**
  - **Evaluate (even? n)**
  - **if result is true, evaluate (/ n 2) and return the result**
  - **otherwise evaluate (/ (- n 1) 2) and return the result**
- **(if (zero? x) 1 (/ 1 x))**

cond

- **(cond (<bool 1> <expr 1>)**
  …
  (<bool n-1> <expr n-1>)
  (else <expr n>))
- **Evaluate bools until one returns true, eval & return corresponding expr, otherwise return value of <expr n>**

```
(cond ((= x 0)      'zilch)
      ((<= x 1)     'one)
      (else         (- 4 2)))
```

not

- `(not #f) => #t, (not #t) => #f`
  - `(not ...)` **is not a special form**

or

- `(or <expr1> … <exprn>)`
  - Evaluate exprs 1 thru n until one returns true
  - If an expr returns a true value, then `or` returns that value
    Otherwise `or` returns false
  - (or …) is a special form

- eg (or (null? x)
            (car x))

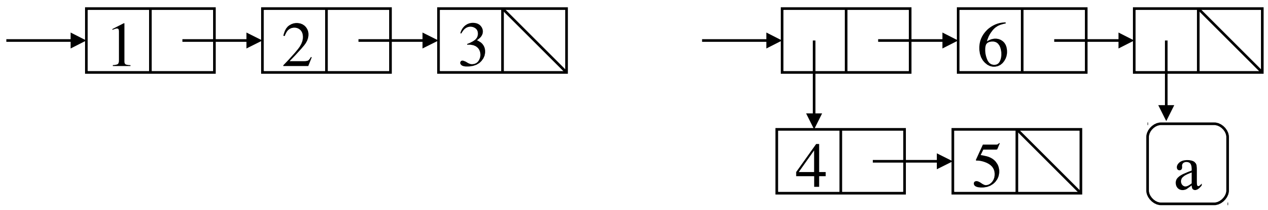
  is roughly the same as
      (if (null? x) (null?x) (car x))

and

- `(and <expr1> … <exprn>)`
  - **Evaluate exprs 1 thru n until one returns #f**
  - **If an expr returns #f, so does `and`**
  - **otherwise, `and` returns value of <exprn>**
  - **(and …) is a special form**

DataTypes: Lists

- Scheme data types:
  - Lists
  - Symbols
  - Numbers
  - Etc.
- External representation: (1 2 3) or ((4 5) 6 a)
  - elements, separated by whitespace, surrounded by ( )
- Internal representation: singly-linked list



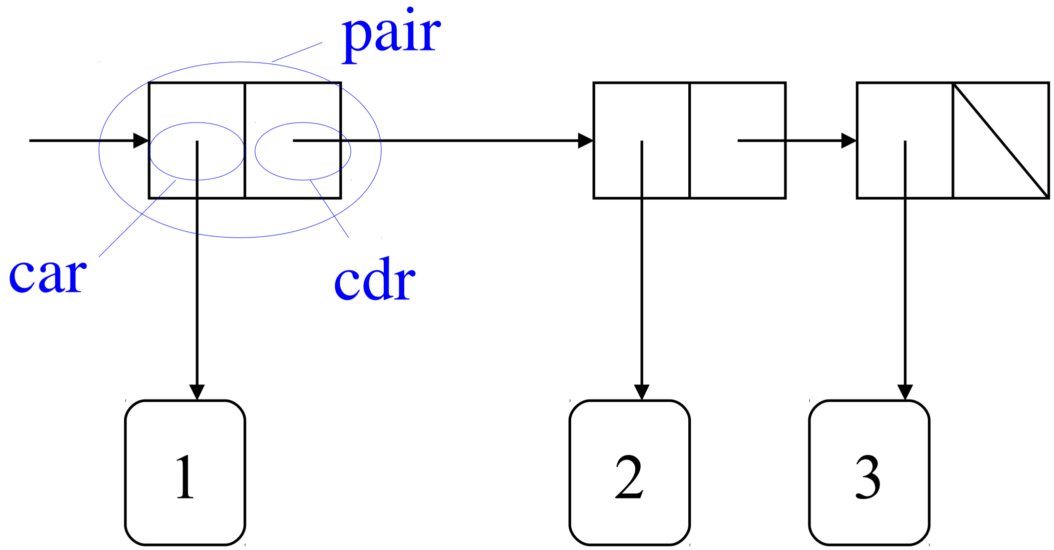
Programs vs data

- **Programs are lists**
- **Lists are data**
- **Programs are data**

- **Question: Is (and b c) a program or data?**

- **Answer: Yes, it is a program or it is data**

Scheme: Lists



- **(car '(1 2 3)) => 1**
- **(cdr '(1 2 3)) => (2 3)**
- **(cons '1 '(2 3)) => (1 2 3)**

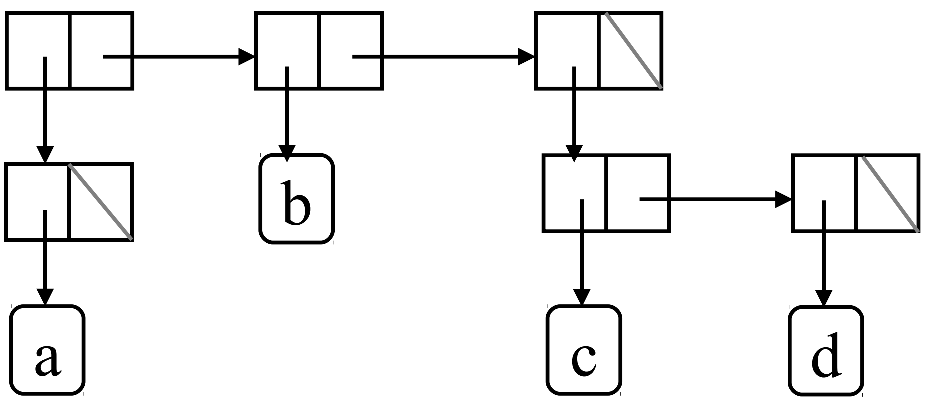
Lists of Lists

Examples:

(car '((a) b (c d))) => (a)

(car (car '((a) b (c d)))) => a

(car (car (car '((a) b (c d))))) => *error*



((a) b (c d))

CONStructing Lists

Examples:

(cons ‘(a b c) ‘((a) b (c d))) => ((a b c) (a) b (c d))

(cons ‘d ‘(e)) => (d e)

(cons ‘(a b) ‘(c d)) => ((a b) c d)

(cons ‘a (cons ‘b (cons 'c '( )))) => (a b c)

C (a|d)+ r

- (cadr x) means (car (cdr x))

  (cadr '(a b c)) => b
- (cdadr x) means (cdr (car (cdr x)))

  (cdadr '(a (b c d) e)) => (c d)
- (cadadr x) means (car (cdr (car (cdr x))))
  - But if you use it you are probably doing something wrong
- (cdadadr x) is not defined

C (a|d)+ r

More examples:

(cdr ‘((a) b (c d))) => (b (c d))

(cadr ‘((a) b (c d))) => b

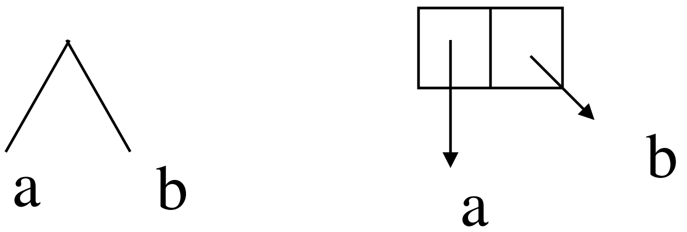
(cddr ‘((a) b (c d))) => ((c d))

(cdddr ‘((a) b (c d))) => ( )

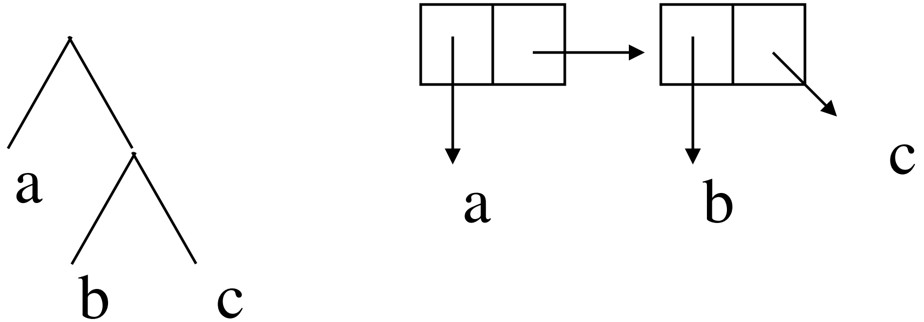
(cddddr ‘((a) b (c d))) => *error*

Improper Lists

(a . b)



(a b . c)



More list operations

- **(null? '( )) => #t**
- **(list 'a (+ 3 4) (cons 'a (cons 'b '( )))) => (a 7 (a b))**
  - **compare with**
    '(a (+ 3 4) (cons 'a (cons 'b '( )))) =>
    (a (+ 3 4) (cons 'a (cons 'b '( ))))
- **(reverse ‘(a b c)) => (c b a)**
  (reverse ‘((a b) c d)) => (d c (a b))
- **(append ‘(a b) ‘(c d)) =>(a b c d)**
  - **compare with (cons '(a b) '(c d)) => ((a b) c d)**
- **(member ‘c ‘(a b c d)) => (c d)**

Dynamic typing

What is the type of the car field of a pair?

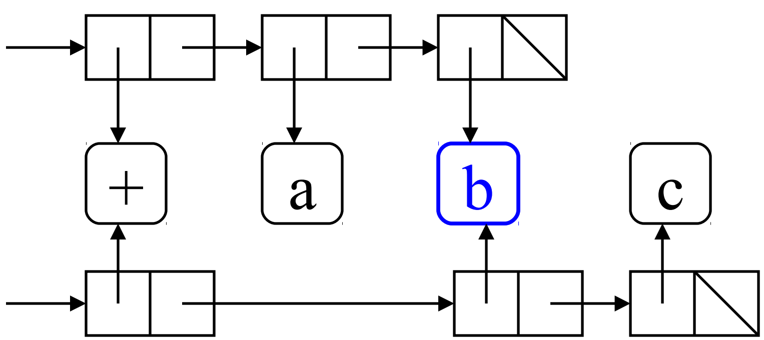
- variables and fields don't have types, only values have types
- A value is represented by a data structure with
  - a type code and
  - a value
- Depending on the type, the value is either
  - immediate data (e.g. integer)
  - a pointer to the actual data in the heap (e.g. a pair)

Data Types: Symbols

- **A, abc, and fish are symbols**
- **So are 3ab and + and a-b**
- **A symbol can be a variable**
- **A symbol can be data**
- **Symbols are "unique-ified" by read: same name always refers to same object in memory**

(+ a b)

(+ b c)



Equality Testing

- (eq? x y) true if x and y are same pointer

(eq? z z)	=> #t	a value is eq? to itself
(eq? 'a 'a)	=> #t	symbols are unique
(eq? 1.0 (+ 0.5 0.5))	=> #f	different locations in heap. test numbers with =

- (= 1.0 (+ 0.5 0.5)) => #t

- (equal? x y): all of above are equal?
  - roughly, x and y are equal? if they print the same

Assoc-lists

- **Assoc-list: a list of (key value) lists**
  - **Eg ((horse 4)(bird 2)(monkey (2 4))**
- **Function (assoc key a-list)**
- **(assoc ‘horse ‘((horse 4)(bird 2)(monkey (2 4))**
  =>(horse 4)
- **(assoc ‘monkey ‘((horse 4)(bird 2)(monkey (2 4))**
  => (monkey (2 4))