Functional Programming with Lists and Strings using Scheme

Bob Keller

January 2010

Functions

- By a function f: A→B, we mean that f determines, for each x ∈ A exactly one y ∈ A. The y is typically written f(x).
- However, in Scheme it is written (f x).
- A is called the **domain** of f
- B is called the **co-domain** of f
- { f(x) | x ∈ A } is a subset of B called the **range** of f.

Application

- When a function is applied to a domain element in Scheme, this is called an **application**:

```
>(cos pi)
-1.0
```

These boxes show Scheme interactions.

- The act of finding the corresponding co-domain element is called **evaluation**.
- The Scheme read-eval-print loop evaluates applications.

Sets of Pairs

- A function f: A→B can be envisioned as a set of pairs of the form (x, y) where $x \in A$ and $y \in B$.

- The set of pairs is sometimes called **the graph** of the function.

Arity

- The “arity” of a function is its number of arguments.
- Technically a function of arity > 1 has the form:

  $f: A^n \rightarrow B$

  where A^n means the set of all **n-tuples** of values from A. This f is called an n–ary function, and n is its arity.

Multi-Arity Functions

- In Scheme, some function names are overloaded to represent more than one arity.
- e.g. + and * represent functions of arbitrary arity:

```
> (+ 4 5)                ;; 2-ary
9
> (+ 4 5 6)              ;; 3-ary
15
> (+ 4);                 ;; 1-ary
4
> (+)                     ;; 0-ary
0
```

```
> (* 4 5)
20
> (* 4 5 6)
120
> (* 4)
4
> (*)
1
```

Use of Parentheses

- Note that in a function application, exactly one set of parens is used for a given application.
- It is incorrect to add other parens for grouping. These represent applications in their own right.
- Furthermore, there is no need for further grouping.

Use of Parentheses

- (+ 2 3 4) **ok**
- (+ (2)(3)(4)) **wrong**.
  2 is not a function being applied.
- (+ (* 2 3) 4) **ok** (* 2 3) is an inner application
- ((2)) **wrong** 2 is not a function;
  neither is (2)

Schme Domains

- The domain of Scheme functions includes:
  - Integers, of arbitrary precision
  - Rationals
  - Floating Point
  - Complex combinations of the above
  - Symbols
  - Strings
  - Booleans (true is #t false is #f)
  - Characters
  - and more

Dynamic Typed Variables

- In Scheme, a **variable** does not have a specific type of value; the value can be any type.
- However, some functions require values of a specific type.
- For example, + expects numbers, and doesn't know what to do with strings.

Type Error Example

```
> (+ 2 3)                    ; ok
5

> (+ "foo" "bar")            ; bad
error +: expects type <number> as 1st argument,
given: "foo"; other arguments were: "bar"
```

The Drawback of a Dynamically-Typed Language

- The compiler typically won't tell you when you are making a type error.
- You won't find out until run-time.
- Depending on the data with which you test, you might not find out until a **very inopportune time** [e.g. the spacecraft is launching].

Boolean Functions

- Boolean functions are generalized to treat anything other than #f as if true.
- **and** is arbitrary-arity and returns #f if some argument is #f, otherwise the last argument.
- **or** returns the first non-false argument, or #f if all are #f.

```
> (and #t #f)
#f
> (and)
#t
> (and #t 99)
99
> (and #t pi)
3.141592653589793
```

```
> (or #t #f)
#t
> (or pi #f)
3.141592653589793
> (or)
#f
> (or #t pi)
#t
```

```
> (not #t)
#f
> (not #f)
#t
> (not pi)
#f
```

Conditional

- The conditional if behaves like a 3-ary function, exception that only one of the 2nd or 3rd argument is evaluated, depending on whether the first is not #f.

```
> (if (< 2 3) 4 5)
4
> (if (< 4 3) 5 6)
6
```

Single Symbols

- Symbols look somewhat like strings
- They are identified by a single **unmatched** quote mark in front of some letters, such as:

  'apple

  'x

  'the_way_it_is

- Caution: There may be multiple characters in the character set that look like a single quote. Only one of them typically works.

Strings

- Strings used **matched** double quote marks.

  "apple"

  "x"

  "the_way_it_is"

- Caution: There may be multiple characters in the character set that look like a double quote. Only one of them typically works.

Symbols vs. Strings

- At most **one instance** of a symbol is ever stored in Scheme.
- Comparison of symbols for equality is done by **address**, rather than content, so it is fast.
- Comparison of strings is by content. It is slow if the string is long.

Symbol to String Conversion

- There are built-in functions:

  symbol->string

  string->symbol

```
> (symbol->string 'apple)
"apple"

> (string->symbol "apple")
apple
```

Note that the quote does not show when a symbol prints.

Evaluating Symbols and Strings

- Symbols and strings evaluate to themselves.

Characters

- Strings are composed of characters.
- **Single characters** are designated by preceding the character with

  #\

  e.g. #\a #\b #\c
- You may also use a single quote **as if** these were symbols, but they really aren't,

  e.g. '#\a '#\b '#\c
- Characters print with the #\.

Special Characters

- Certain keywords are used to identify special characters:

#\space	space character
#\newline	new line character
#\tab	tab character
#\(	left paren character
#\)	right paren char character

Lists

- The **list abstraction** is the workhorse of Scheme.
- It is a universal method for structuring data.
- In CS 60, we call this the “open list” abstraction.
- It is also used in other functional and logic languages, such as Lisp, Prolog, Haskell, etc.

List Depiction

- Lists are shown with parentheses around them for grouping purposes:

  (This is a list of symbols)

  (so is this)

Programs as Lists

- Note that Scheme function definitions look a lot like lists.
- This is intentional.
- Later we will see how we can build our own language easily using this idea.

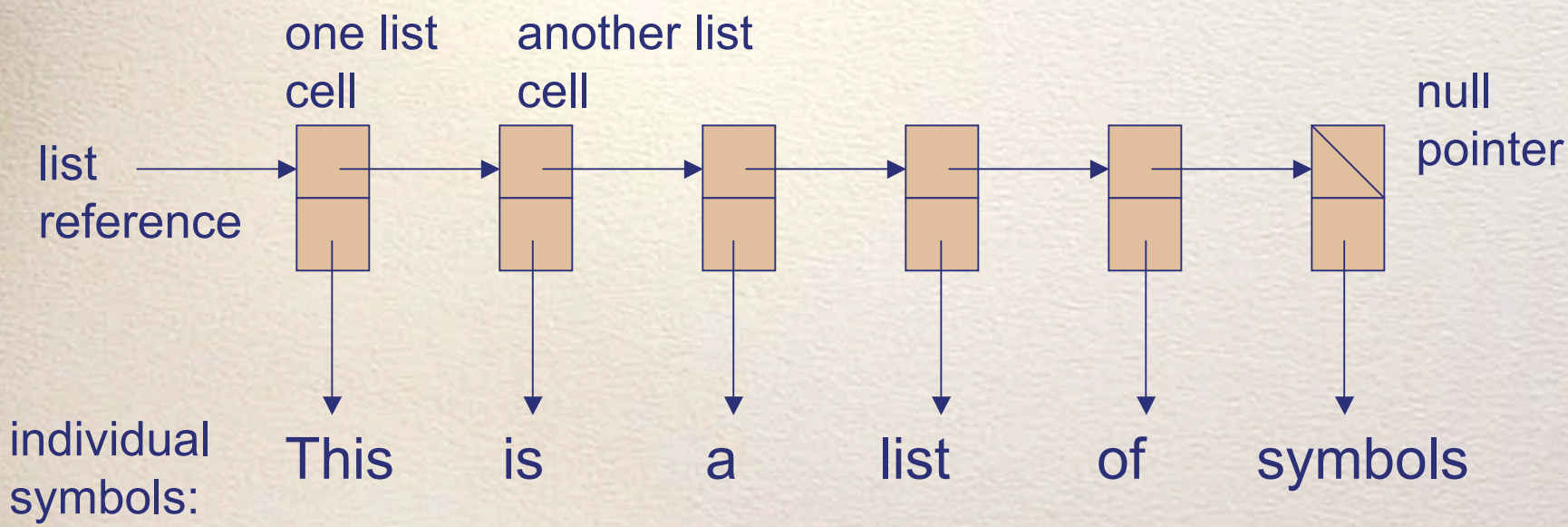
Literal Lists

- A **literal** list (a list of constant values in code) is indicated by a **single quote**:

  '(This is a list of symbols)

- The list above is **not a string or symbol** once entered into Scheme. When the compiler reads it, it strips off the parens and stores each item separately, so that we can rapidly move from item to item.

List Representation in Memory



A list is identified by a pointer to its first cell.

Empty List

- The empty list is designated by either

  ()

  or '()

Mixed Lists

- The elements of a list do not have to be all of the same type. For example, symbols and numbers can be mixed:

'(This is a list containing 2 numbers 4567)

Creating non-Literal Lists

- The built-in function ***list*** creates a list from an arbitrary number of elements.
- Its arguments are first **evaluated**, then the list is created.

```
>  (list '(The answers are) (+ 2 3) (* 4 5))
((The answers are) 5 20)
> (list 'one)
(one)
> (list)
()
```

Reversing a list

- The built-in function reverse creates a new list with the elements of the original in reverse order.

```
> (reverse '(1 2 3 4 5))
(5 4 3 2 1)
```

Appending lists

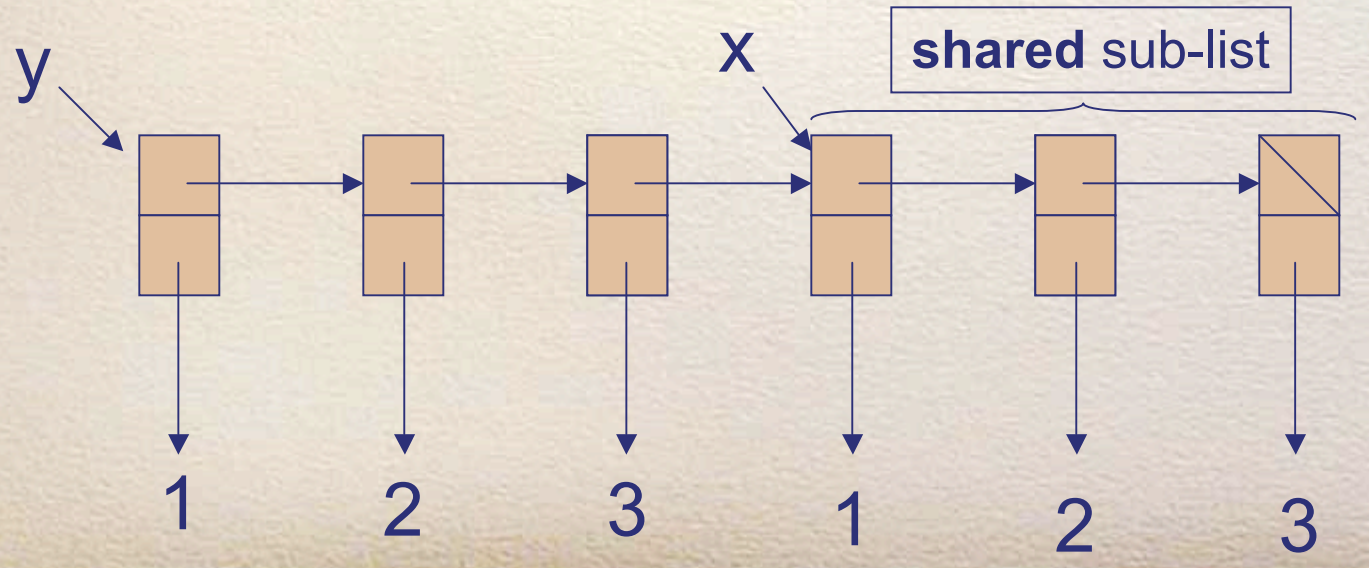
- Appending lists constructs a new list having the elements of the argument lists in the order given.
- Append has arbitrary arity

```
> (append '(1 2 3) '(4 5))
(1 2 3 4 5)
> (append '(1 2) '(3 4) '(5 6))
(1 2 3 4 5 6)
> (append '(1 2 3))
(1 2 3)
> (append)
()
```

List Sharing

- Scheme will freely share sub-lists when possible.

```
> (define x '(1 2 3))
> x
(1 2 3)
> (define y (append x x))
(1 2 3 1 2 3)
```



List Sharing

- Sharability of lists goes hand-in-hand with functional programming.
- Shared lists should not be modified in place, lest there be rude surprises.
- Adhering to functional style prevents such modifications.

Built-in Functions on Lists

- **empty?** is used to tell whether or not the argument list is empty
- **length** returns the length of the list.
- Prefer **empty?** to computing the **length** of a list and comparing it to 0. It takes time to compute the length.

Getting Characters of a String

- **string->list** returns a list of characters in its string argument
- **list->string** does the opposite

```
> (string->list "Don't worry!")
(#\D #\o #\n #\' #\t #\space #\w #\o #\r #\r #\y #\!)

> (list->string (list #\D #\o #\n #\' #\t #\space #\w #\o #\r #\r #\y #\!))
"Don't worry!"
```

Appending Strings

- string-append is a useful built-in

```
> (string-append "the " "rain " "in " "Spain")
"the rain in Spain"
```

Lists with Lists as Elements

- Lists can have other lists as elements, those lists can have lists as elements, etc.
- Only one outer quote is needed to capture all of the nesting structure in a literal list:

```
'(This is (a list) containing (2 numbers) 4567)
```

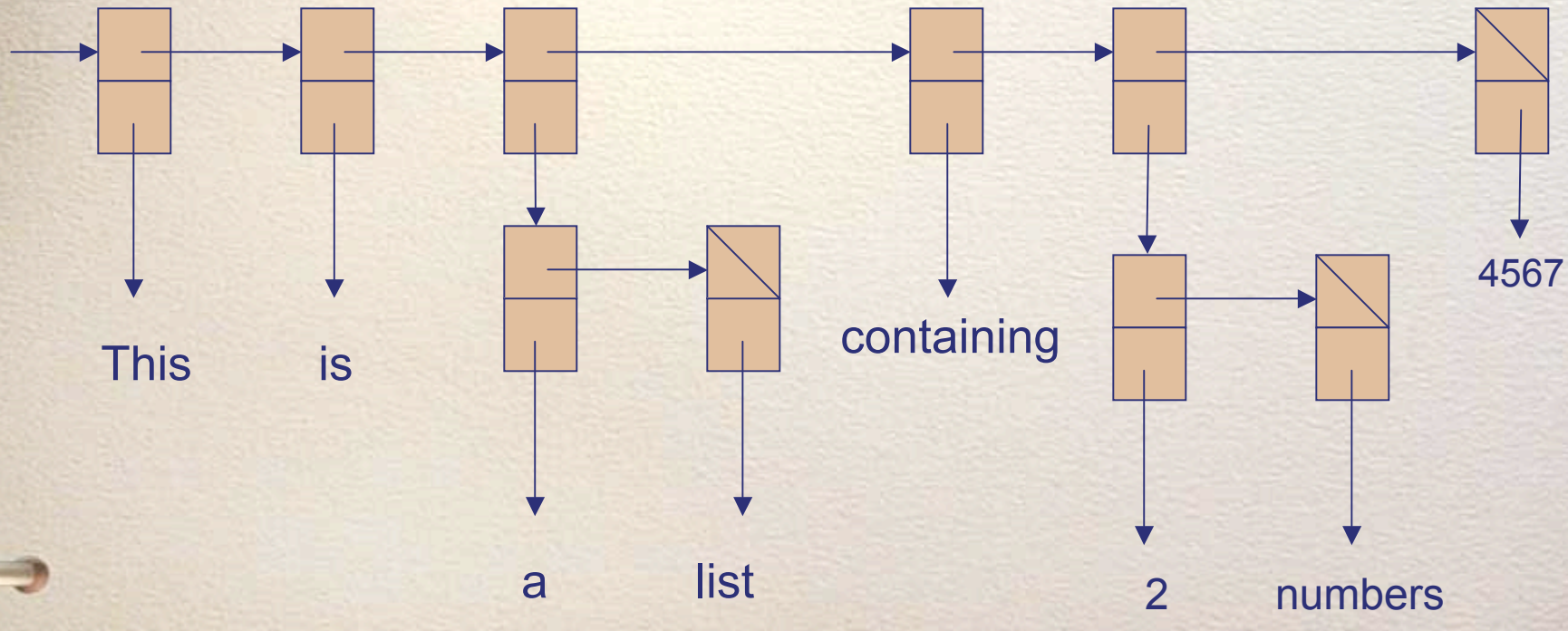
Number of Elements

- The number of elements in a list is regarded as the number of outermost items.
- The following list has 6 elements, not 8:

```
'(This is (a list) containing (2 numbers) 4567)
```

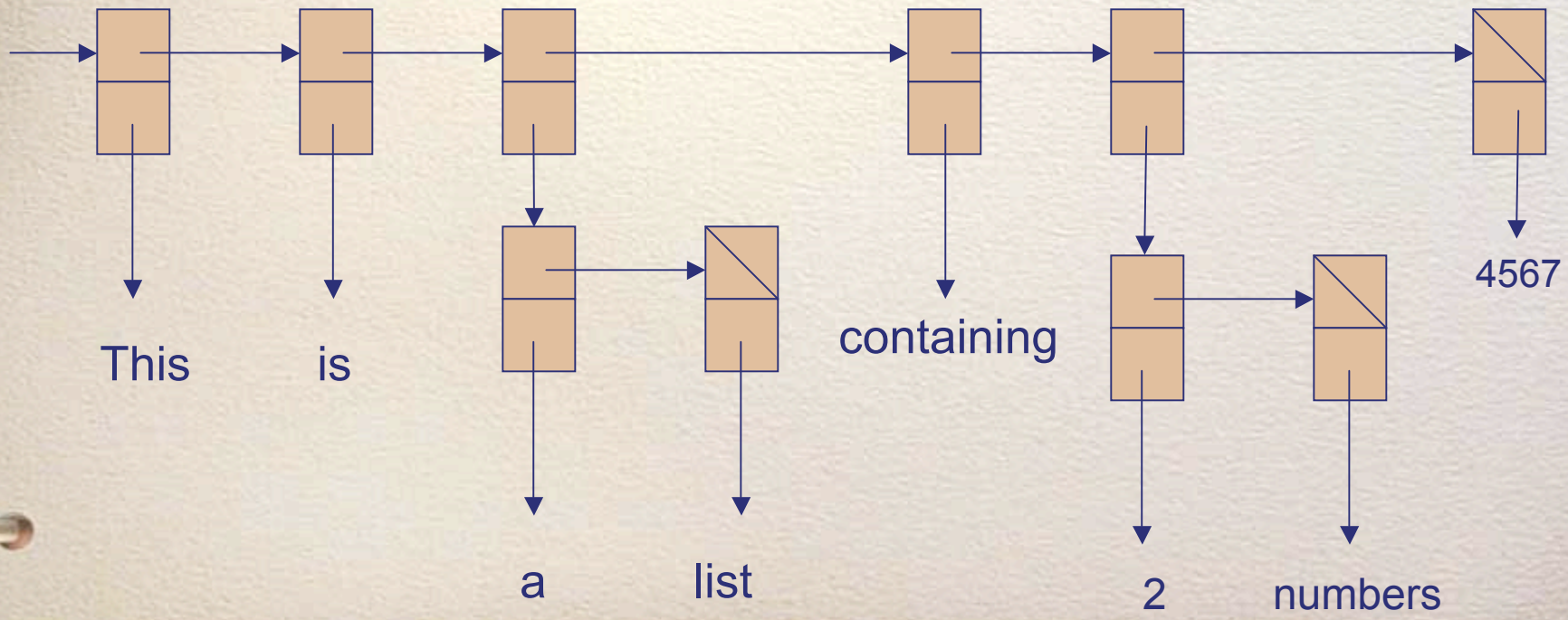
Representing Nested Lists

'(This is (a list) containing (2 numbers) 4567)

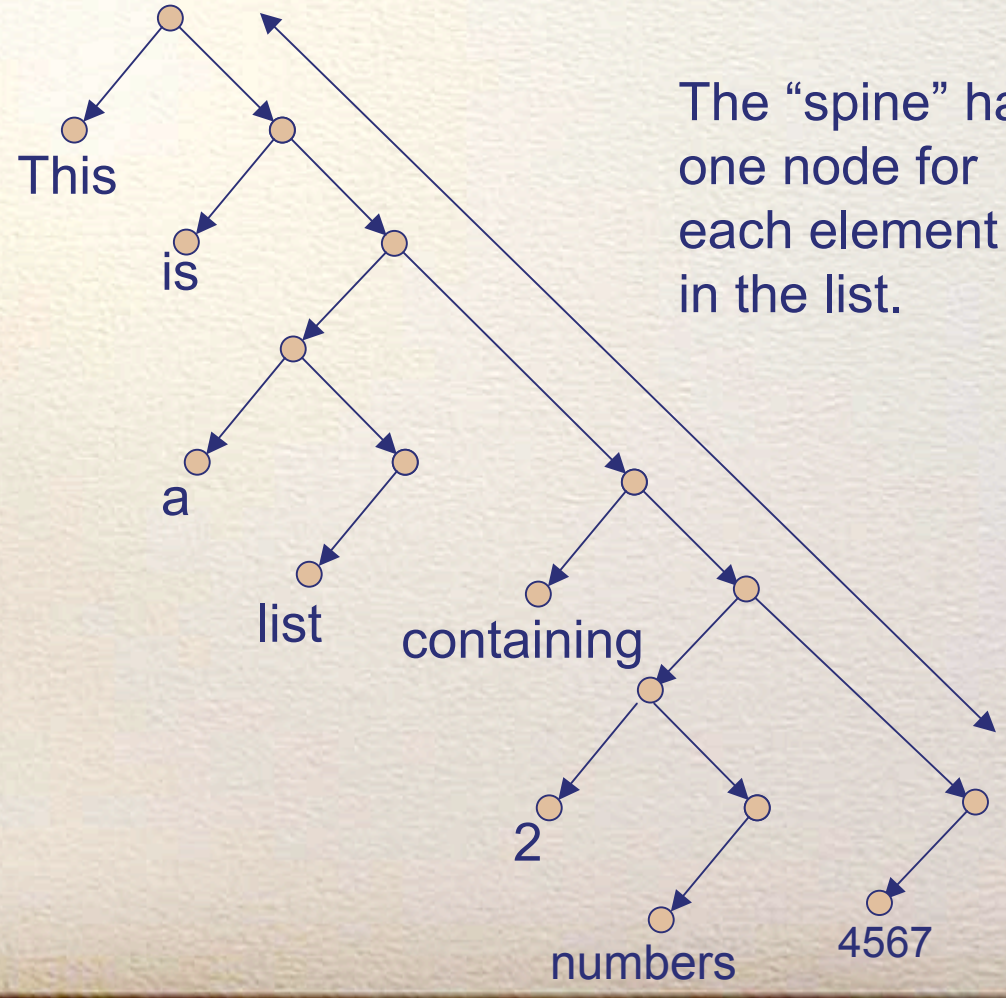


Lists as Trees

- If no elements of a list are shared, then the list is essentially represented as a **tree**:



Lists as Trees



The “spine” has one node for each element in the list.

Accessing List Elements

- Before accessing elements of a list, it should be determined that the list is non-empty.
- **first** returns the first element of a non-empty list
- **rest** returns the list of all but the first element of a non-empty list.
- Both have a **constant-time** cost [O(1) cost].

Accessing list elements

```
> (define x '(1 2 3))
> (first x)
1
> (rest x)
(2 3)
> (first (rest x))
2
> (first (rest (rest x)))
3
> (rest (rest (rest x)))
()
> (rest (rest (rest (rest x))))
error rest: expected argument of type <non-empty list>; given ()
>
```

Convenience Functions

```
> (define x '(1 2 3 4 5 6 7 8 9))
> (first x)
1
> (second x)
2
> (third x)
3
> (eighth x)
8
> (ninth x)
error reference to undefined identifier: ninth
```

Asymmetry

- Lists are intentionally asymmetric, to provide a simple, elegant implementation.
- Consequently, there is no built-in **last** function.
- The **business-end** of a list is at its first item.

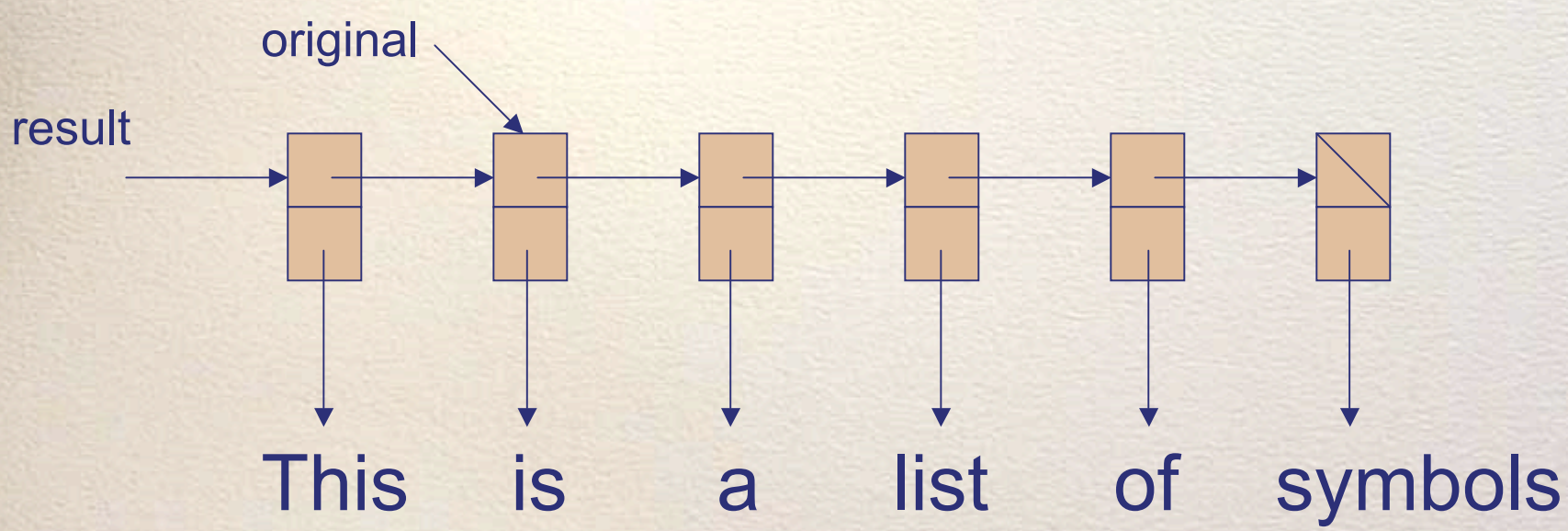
cons'ing to a list

- cons constructs a new list from an existing list, by adding a new element to the front.
- The original list is ***shared*** with the new list.

```
> (cons 0 '(1 2 3))
(0 1 2 3)
```

cons'ing illustrated

- (cons 'This '(is a list of symbols))



Recall that a list is identified by a pointer to its first element.

cons vs. append

- **cons** expects an **element** and a list
- **append** expects (typically) two lists, although it can take any number

```
> (cons 0 '(1 2 3))
(0 1 2 3)

> (append 0 '(1 2 3))
error append: expected argument of type <proper list>; given 0
```

cons vs. append

- **cons** expects an element and a list.
- **append** expects (typically) two lists.
- Be careful, because **in some cases, either makes sense**, but will give different results:

```
> (append '(0 1) '(2 3))
(0 1 2 3)

> (cons '(0 1) '(2 3))
((0 1) 2 3)
```

append, list, cons distinction

```
> (append '(1 2 3) '(4 5))
(1 2 3 4 5)

> (list '(1 2 3) '(4 5))
((1 2 3) (4 5))

> (cons '(1 2 3) '(4 5))
((1 2 3) 4 5)

> (list '(1 2 3) '(4 5) '() '(6 7))
((1 2 3) (4 5) () (6 7))

> (append '(1 2 3) '(4 5) '() '(6 7))
(1 2 3 4 5 6 7)
```

Using Types to Clarify

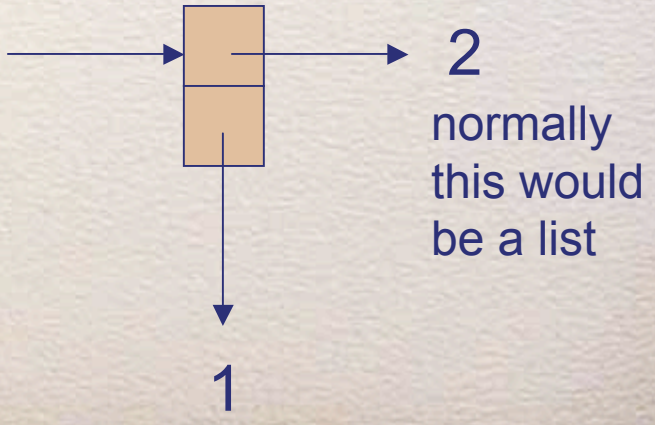
- Suppose A is a data type.
- By A* let's mean the type **list of A's**.
- The type of 2-argument **append** is:
  append: A* x A* → A*.
- The type of **cons** is:
  cons: A x A* → A*
- The type of 2-argument function **list** is:
  list: A x A → A*

Improper use of cons

- cons will happliy accept arguments of type A x A and return a “dotted pair”.

- Using this facility is **not advised**.

```
> (cons 1 2)
(1 . 2)
```



Accessing by Number

- **list-ref** returns the nth element of a list, where elements are indexed starting at 0. The first argument is a list, and the second an index.

- Note: **Don't use list-ref to iterate** over a list. It is too **slow**. The cost of access is **linear**, not constant, in the value of the index [O(n) cost, where n is the index].

```
> (list-ref '(a b c d e) 3) ;; a is element 0
d
```

Association Lists (A-Lists)

- Association lists are a special variety of nested list. Its elements are lists of two elements.

- They are used for representing functions by enumerating the graph of the function.

A-List Example

- Let's associate the first name with the last names of some notable presidents:

  '((Washington George)
  (Lincoln Abraham)
  (Jefferson Thomas)
  (Obama Barack))

assoc Function

- The assoc function is designed to look up on the first items of an association list.
- It returns the entire list of what it finds.
- If it finds nothing, #f is returned.

```
> (assoc 'Jefferson '((Washington George) (Lincoln Abraham) (Jefferson Thomas) (Obama Barack)))
(Jefferson Thomas)

> (assoc 'Obama '((Washington George) (Lincoln Abraham) (Jefferson Thomas) (Obama Barack)))
(Obama Barack)

>  (assoc 'Bush '((Washington George) (Lincoln Abraham) (Jefferson Thomas) (Obama Barack)))
#f
```

Idiomatic Usage of assoc

- Use the facts that
  - assoc returns #f if it finds no match.
  - anything other than #f is interpreted as true.

```
(define presidents '((Washington George) (Lincoln Abraham)
                     (Jefferson Thomas) (Obama Barack)))

(define (get-first-name last-name)
  (let (
        (found (assoc last-name presidents))
        )
    (if found
        (second found)
        (error "last name not found: " last-name))))
```

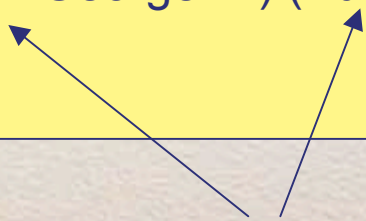
Run of previous code

```
> (get-first-name 'Lincoln)
Abraham

> (get-first-name 'Obama)
Barack

> (get-first-name 'Bush)
error last name not found:  Bush
```

assoc Function

- If there are **multiple matches** in the list, only the first is returned.
- So assoc **implements a function**, even if the A-list is not strictly the graph of a function.

```
> (assoc 'Bush '((Obama Barack) (Bush George W) (Bush George)))

(Bush George W)
```



ambiguous entries

member function

- To find whether or not an element occurs in a list …
- (member Element List) returns the **first suffix** of List beginning with Element, or #f if Element does not occur in List.

```
> (member 4 '(1 2 3 4 5 6))
(4 5 6)
> (member 7 '(1 2 3 4 5 6))
#f
```

Sorting a List

- Lists can be sorted in many ways, especially if the elements are themselves lists.

- The built-in function **sort** requires two arguments:
  - a list of things to be sorted
  - a **comparison function** that compares two items in a list and returns #t if the first is “less than” the second, and #f otherwise

Sorting Examples

```
> (sort '(5 7 3 1 8 9 2) <)
(1 2 3 5 7 8 9)

>  (sort '(5 7 3 1 8 9 2) >)
(9 8 7 5 3 2 1)

> (list->string (sort (string->list "what's the matter?") char<?))
"  '?aaeehhmrsttttw”
;Note: The first two characters are spaces.
```

```
(define (compare-last-names x y)
    (string<? (symbol->string (first x)) (symbol->string (first y))))

(sort '((Obama Barack) (Bush George W) (Bush George)) compare-last-names)
```

Equality Testing

- **equal?** is useful for **general equality** testing, including lists, numbers, symbols, etc.
- **=** only applies to **numbers**
- **string=?**, **char=?**, **symbol=?** apply to their respective types
- **eq?** tests for **same location in memory** and **should not be used casually**

Equality Examples

```
> (equal? 'foo 'bar)
#f
> (= 'foo 'bar)
error =: expects type <number> as 1st argument, given: foo
> (symbol=? 'foo 'bar)
#f

> (eq? "foo" "foo")
#f
> (eq? 'foo 'foo)
#t
```

Anagram Example

- Are two strings anagrams of one another (have the same letters)?

```
> (anagram? "bob keller" "broke bell")
#t
```

```
(define (sort-chars x) (sort (string->list x) char<?))

(define (anagram? x y) (equal? (sort-chars x) (sort-chars y)))
```

In this example, spaces “count” as characters.