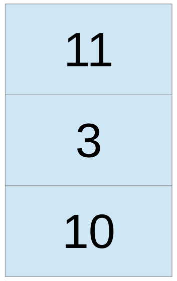
Representing data types with lists

- Lists will be central to most of our lisp programming, but we need some way to represent structured data (trees, graphs, stacks, etc)
- As long as we choose an appropriate structure and tailor our functions to operate on that, we can represent nearly any desired data type through nested lists
- The code to manipulate the data is often quite elegant, but the raw data is often not terribly human readable

Stack example

- The stack is a very simple ADT to represent with our lists
- cons and car operate on the front of the list, and stack pop and push operate on the top of the stack, so it makes sense to represent our stack as a list whose “top” is the front of the list and bottom is the end of the list



‘(11 3 10)

Simple stack implementation

- Create a new stack, optionally specify starting content

```
(defun stack (&optional (S nil)) (if (stackp S) S nil))
```

- Check if S is a stack (really just see if it’s a list)

```
(defun stackp(S) (listp S))
```

- Return the current size of the stack

```
(defun stSize(S) (if (stackp S)) (length S) ‘error))
```

- Push an element onto the stack

```
(defun push(S e) (if (stackp S) (cons e S) ‘error)
```

Record representation

- Suppose we want to represent a student as a record with several fields: name, student number, email address
- use a list of three string elements: name, stnum, email
- create functions to create/use/manipulate these
  - generate and return a new student record from params

    ```
    (defun studentRec (name stnum email) .... )
    ```

  - check if something is a valid student record (valid-looking name, student number, email address)

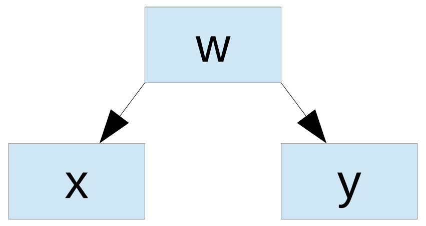
    ```
    (defun studentRecp (S) ....)
    ```

Binary tree representation

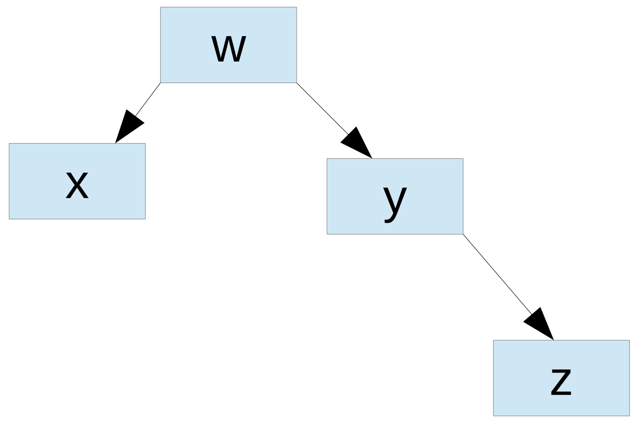
- Suppose we wish to represent binary trees
- We could choose to represent each node in the tree as a 3-element list (data leftsubtree rightsubtree), using nil for an empty tree or subtree
- An empty tree would simply be ‘()
- tree with 1 in the root, no subtrees would be ‘(1 nil nil)
- Tree with 10 in root, 5 in left subtree, no right subtree would be ‘(10 (5 nil nil) nil)

Binary tree layout example

(w (x nil nil) (y nil nil))



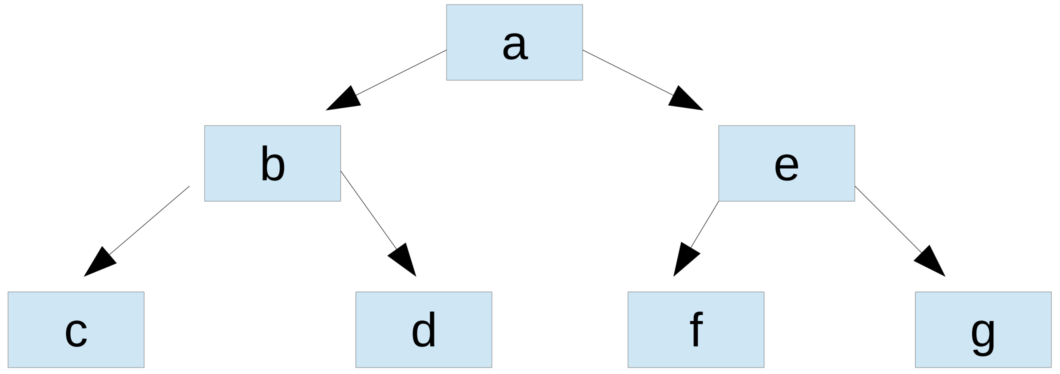
(w (x nil nil) (y nil (z nil nil)))



Binary tree example cont.

- (a (b (c nil nil) (d nil nil)) (e (f nil nil) (g nil nil)))

(as we mentioned, raw data not very human friendly)



Sample binary tree methods

- Create a new tree from a data value

```
(defun tree (d) (list d nil nil))
```

- Get the left subtree of a tree

```
(defun getLeft (tr)
   (if (and (treep tr) (not (null tr))) (cadr tr) ‘error))
```

- Insert L as the left subtree of tr, fail if already in use

```
(defun setLeft (tr L)
   (if (and (treep tr) (treep L) (null (getLeft tr)))
       (setf (cadr tr) L) ‘error))
```

tree methods cont.

- Check if something is a valid tree (every node in it has a data field and two valid subtrees)

```
(defun treep (tr)
   (cond
      ((not (listp tr)) nil)
      ((null tr) t)
      ((/= 3 (length tr)) nil)
      ((not (treep (getLeft tr))) nil)
      ((not (treep (getRight tr))) nil)
      (t t)))
```

and more tree methods ...

- Printing the tree content: traversals

```
(defun printTree (tr)
   (cond
     ((not (treep tr)) (format t “Error: invalid tree~%”))
     ((null tr) t)
     (t (block ‘InorderTraversal
            (printTree (getLeft tr))
            (format t “~A~%” (getData tr))
            (printTree (getRight tr))))))
```