Lisp Programming

Boolean Functions

- Return T for true and nil for false
  - **(atom item)**
    - **is item an atom, i.e. not a cons cell**
  - **(listp item)**
    - **is item a list, i.e. a cons cell**
  - **(null item)**
    - **is item the empty list nil**
- In general have a predicate for every type
  - **e.g. numberp, listp**

Logical Operators

- Reverse a boolean result
  - » **(not (atom item))**
- Combine predicates – lazy evaluation
  - » **(and (listp a) (listp b))**
    - > **stop evaluating as soon as false is found**
  - » **(or (listp a) (listp b))**
    - > **stop evaluating as soon as true is found**
  - » **(and (listp a) (listp (car a)))**
    - > **If a is not a list then (car a) would fail**
    - > **lazy evaluation prevents this**

Conditonal – cond

- General template
  - » (cond (p.1 s.1-1 s.1-2 ... s.1-p)
    (p.2 s.2-1 s.2-2 ... s.2-q)
    ...
    (p.n s.n-1 s.n-2 ... s.n-r)
)
  - » p.i are predicates
  - » s.i-k is the k'th S-expression for predicate p.i
    - > Usually only one per predicate

Conditonal – cond – 2

- Uses lazy evaluation
  - » Evaulate p.i in turn, for i : 1 .. n
  - » For the first true p.i evaluate s.i-1 ... s.i-r
    - > Value of cond is value of s.i-r
  - » If all p.i are false, value of cond is nil
- Example
    - > note use of t ∫ true to handle the otherwise case
  - » (cond ((atom a) a)
    ((atom (car a)) (print (car a)) (cdr a))
    (t (process (car a)))
)

Recursion

- Only looping method in **pure** Lisp is recursion
- In general, recursion involves
  - **taking a list apart**
    - **(car theList) (cdr theList)**
  - **doing recursion on the parts**
    - **(recursiveCall (car theList))**
      (recursiveCall (cdr theList))
  - **rebuilding a new list from the parts of the old list**
    - **(cons (recursiveCall (car theList))**
      (recursiveCall (cdr theList)))
  - **empty list is often used used for termination**
    - **(cond ((null theList) (do base case)) ...)**

Recursion a general template

```
( defun recursive (theList otherParameters)
  (cond ( (null theList) (first base case) )
            ...
          ( (pred1)  (last base case) )
          ( (pred2)  (first nonbase case) )
           ...
          ( t  (last nonbase case) )
))
```

Recursion example 1

- Remove **item** from **list** only at the top level

```
(defun removeTop (list item)
 (cond ( ( null list )  nil )
       ( ( equal  ( car list )  item )
            ( removeTop  ( cdr list )  item ) )
       ( t  ( cons  ( car list )
                    ( removeTop  ( cdr list )  item ) ) )
))
```

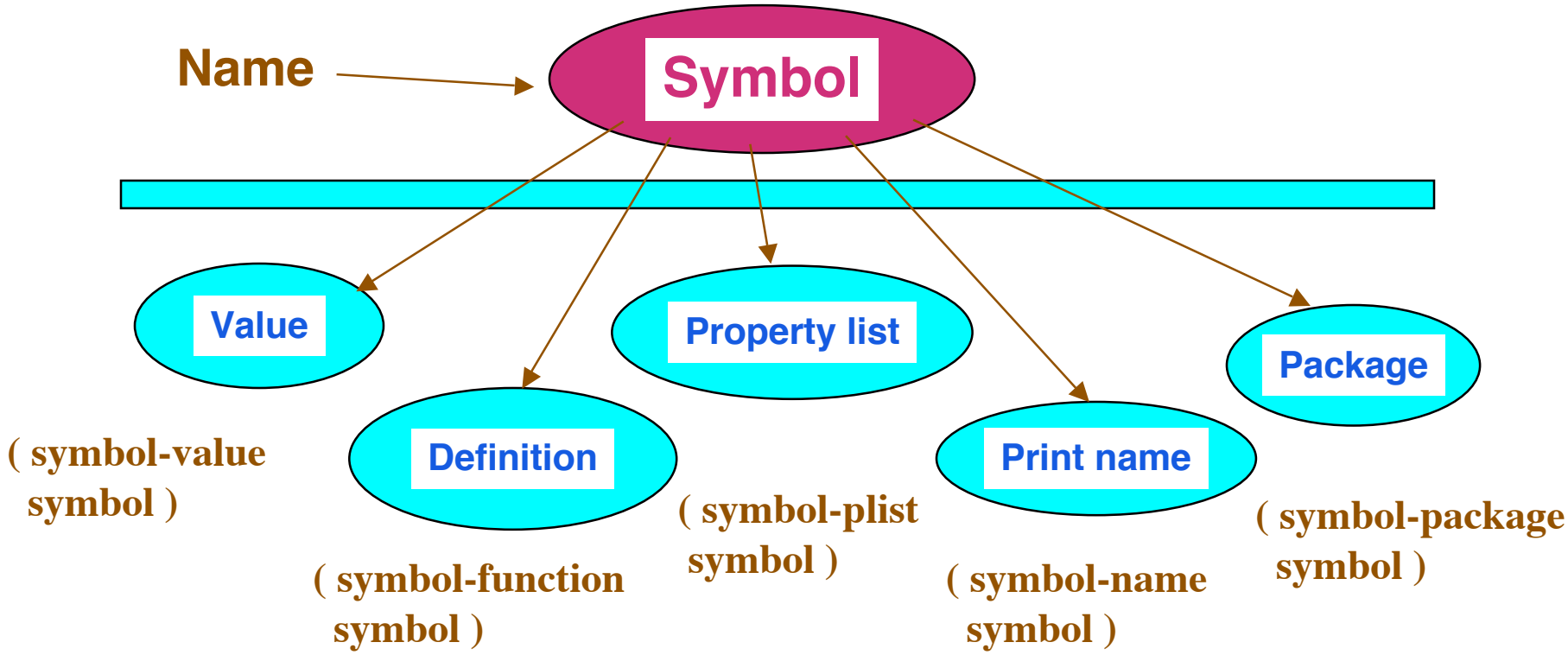
Recursion example 2

◊ Remove **item** from **list** from all levels

```
(defun removeAll (list item)
 (cond ( ( null list )  nil )
       ( ( equal  ( car list )  item )
            ( removeAll  ( cdr list )  item ) )
       ( ( atom  ( car list ) )
            ( cons  ( car list )
                 ( removeAll  ( cdr list ) item ) ) )
       ( t  ( cons  ( removeAll  ( car list ) item )
                 ( removeAll  ( cdr list ) item ) ) )
))
```

Symbols are more than variables

They have a complex structure – See notes on symbols



Access Functions

From Harrison pages 30-38

What is a Property List?

- Programs model the world as we see it
- Entities with attributes (properties in Lisp) populate the world
  - » **entity book with attributes author, publisher, number of pages**
  - » **attributes have values**
    - > **author Asimov publisher ACE books pages 412**
- Lisp models the above with
  - » **symbol book**
  - » **property list**
    (author Asimov publisher ACE books pages 412)

EBNF definition of Property Lists

- Property lists are **attribute-value** pairs all at the same list level.
- EBNF for property Lists

```
PropertyList ::= ( nil , PropName PropValue
                                  PropertyList ) ;
PropName ::= any symbol ;   // name of the property
PropValue ::= any S-expression ;  // its value
```

  - **Examples**

```
> ( colour red size large )
> ( colour white
    change (penny 3 dime 4 looney 6 toonie 10) )
```

  - **Values can themselves be property lists**

Accessing properties-1

◊ Use (get ‘symbol ‘propName) to access a property value for a symbol

» Assume the symbol purse has the property list

> (colour white
change (penny 3 dime 4 looney 6 toonie 10))

» Then

> (get ‘purse ‘change)

> returns the S-expression
(penny 3 dime 4 looney 6 toonie 10)

Accessing properties-2

- Use (**setf (get symbol propName) value**) to set a property value
  - » **(setf (get ‘purse ‘colour) ‘pink)**
    - > **changes the colour of the purse to pink**
  - » **the get returns the address of where the attribute-value is**
  - » **or would be**
    - > **Hence new attribute-value pairs can be stored**

getf and property lists

- (**getf prop-list ‘propName**) accesses properties as well; the first argument is a property list.

  > (**setf (get ‘purse ‘colour) ‘pink**)

  > (**getf (symbol-plist ‘purse) ‘colour**) — **returns pink**

- Property lists do not need to be associated with a symbol’s plist
  - **Any list of attibute-value pairs will do**

    > (**setq x ‘(colour blue change (penny 4 dime 5))**)

    > (**getf x ‘change**) **returns (penny 4 dime 5)**
  - **Even just a property list structure will do**

    > (**getf ‘(colour blue change (penny 4 dime 5)) ‘change**) **returns (penny 4 dime 5)**

Association lists

- Like property lists
- Associate attributes with values
- Uses lists of lists
  - » ((colour black) (size large))
- First of each sublist is the property (key) and the second is the value.

Property List ambiguity

- If a property does not exist **get** returns **nil**
- What if a property value is nil?

Property list ambiguity 2

Good mathematical and programming practice is to give a special name for nil property values.

Could use **(change none)** in place of **(change nil)**

Then **nil** would mean the property does not exists as opposed to the value of an existing property is nil