Lisp

Versions of LISP

- **Lisp** is an old language with many variants
- Lisp is alive and well today
- Most modern versions are based on **Common Lisp**
- **LispWorks** is based on Common Lisp
- **Scheme** is one of the major variants
- The essentials haven’t changed much

Recursion

- Recursion is essential in Lisp
- A *recursive definition* is a definition in which
  - certain things are specified as belonging to the category being defined, and
  - a rule or rules are given for building new things in the category from other things already known to be in the category.

Informal Syntax

- An *atom* is either an integer or an identifier.
- A *list* is a left parenthesis, followed by zero or more S-expressions, followed by a right parenthesis.
- An *S-expression* is an atom or a list.
- Example: `(A (B 3) (C) ( ( ) ) )`

Formal Syntax (approximate)

- <S-expression> ::= <atom> | <list>
- <atom> ::= <number> | <identifier>
- <list> ::= (<S-expressions>)
- <S-expressions > ::= <empty>
  | <S-expressions > <S-expression>
- <number> ::= <digit> | <number> <digit>
- <identifier> ::= *string of printable characters, not including parentheses*

T and NIL

- NIL is the name of the empty list
- As a test, NIL means "false"
- T is usually used to mean "true," but…
- …anything that isn't NIL is "true"
- NIL is both an atom and a list
  - it's defined this way, so just accept it

Function calls and data

- A function call is written as a list
  – the first element is the name of the function
  – remaining elements are the arguments
- Example: (F A B)
  – calls function F with arguments A and B
- Data is written as atoms or lists
- Example: (F A B) is a list of three elements
  – Do you see a problem here?

Quoting

- Is (F A B) a call to F, or is it just data?
- All *literal data* must be quoted (atoms, too)
- (QUOTE (F A B)) is the list (F A B)
  – QUOTE is a "special form"
  – The arguments to a special form are not evaluated
- '(F A B) is another way to quote data
  – There is just one single quote at the beginning
  – It quotes one S-expression

Basic Functions

- CAR returns the head of a list
- CDR returns the tail of a list
- CONS inserts a new head into a list
- EQ compares two atoms for equality
- ATOM tests if its argument is an atom

Other useful Functions

- (NULL S) tests if S is the empty list
- (LISTP S) tests if S is a list
- LIST makes a list of its (evaluated) arguments
  - (LIST 'A '(B C) 'D) returns (A (B C) D)
  - (LIST (CDR '(A B)) 'C) returns ((B) C)
- APPEND concatenates two lists
  - (APPEND '(A B) '((X) Y)) returns (A B (X) Y)

CAR

- The CAR of a list is the first thing in the list
- CAR is only defined for *nonempty* lists

If L is	Then (CAR L) is
(A B C)	A
((X Y) Z)	(X Y)
(( ) ( ))	( )
( )	*undefined*

CDR

- The CDR of a list is what's left when you remove the CAR
- CDR is only defined for *nonempty* lists
- The CDR of a list is always a list

CDR examples

If L is	Then (CDR L) is
(A B C)	(B C)
((X Y) Z)	(Z)
(X)	( )
(( ) ( ))	(( ))
( )	*undefined*

CONS

- CONS takes two arguments
  - The first argument can be any S-expression
  - The second argument should be a list
- The result is a new list whose CAR is the first argument and whose CDR is the second
- Just move one parenthesis to get the result:

CONS of A (B C) gives (A B C)

CONS examples

- CONS puts together what CAR and CDR take apart

L	(CAR L)	(CDR L)	(CONS (CAR L) (CDR L))
(A B C)	A	(B C)	(A B C)
((X Y) Z)	(X Y)	(Z)	((X Y) Z)
(X)	X	( )	(X)
(( ) ( ))	( )	(( ))	(( ) ( ))
( )	*undefined*	*undefined*	*undefined*

Dotted Pairs

- The second argument to CONS should be a list
- If it isn't, you get a *dotted pair*
- CONS of A and B is (A . B)
- We aren't using dotted pairs in this class
- If you get a dotted pair, it's because you gave CONS an atom as a second argument

EQ

- EQ tests whether two atoms are equal
  - Integers are a kind of atom
- EQ is undefined for lists
  - it might work for lists, it might not
  - but it won't give you an error message
- As with any predicate, EQ returns either NIL or something that isn't NIL

ATOM

- ATOM takes any S-expression as an argument
- ATOM returns "true" if the argument you gave it is an atom
- As with any predicate, ATOM returns either NIL or something that isn't NIL

COND

- COND implements the if…then…elseif…then…elseif…then… control structure
- The arguments to a function are evaluated before the function is called
  - This isn't what you want for COND
- COND is a *special form*, not a function

Special forms

- A *special form* is like a function, but it evaluates the arguments as it needs them
- COND, QUOTE and DEFUN are special forms
- You can define your own special forms
- We won't be defining special forms in this course

Form of the COND

```
(COND
      (condition1   result1 )
      (condition2   result2 )
      . . .
      (T    resultN ) )
```

Defining Functions

- (DEFUN *function_name parameter_list*
  function_body)
- Example: Test if the argument is the empty list
- (DEFUN NULL (X)

```
   (COND
       (X   NIL)
       (T   T) )   )
```

Example: MEMBER

- As an example we define MEMBER, which tests whether an atom is in a list of atoms
- ```
  (DEFUN MEMBER (A LAT)
      (COND
        ((NULL LAT)   NIL)
        ((EQ A (CAR LAT))   T)
        (T   (MEMBER A (CDR LAT))) ) )
  ```
- MEMBER is typically a built-in function

Rules for Recursion

- Handle the base ("simplest") cases first
- Recur only with a "simpler" case
  - "Simpler" = more like the base case
- Don't alter global variables (you can't anyway with the Lisp functions I've told you about)
- Don't look down into the recursion
  ```

Guidelines for Lisp Functions

- Unless the function is trivial, start with COND.
- Handle the base case first.
- Avoid having more than one base case.
- The base case is usually testing for NULL.
- Do something with the CAR and recur with the CDR.

Example: UNION

```
(DEFUN UNION (SET1 SET2)
    (COND
       ((NULL SET1)   SET2)
       ((MEMBER (CAR SET1) SET2)
         (UNION (CDR SET1) SET2)  )
       (T (CONS (CAR SET1)
            (UNION (CDR SET1) SET2)  ))  )  )
```

Still more useful Functions

- (LENGTH L) returns the length of list L
- (RANDOM N) , where N is an integer, returns a random integer >= 0 and < N.

The End