316 12-03-09

* Lisp:
  + Lisp has 2 major dialects:
    - Common Lisp
    - Scheme
  + We will look at pure Scheme, the functional sublanguage of Scheme – as an example of functional language
  + Pure functional languages – all computation is based on expression evaluation, chain of function calls
    - No loops, no assignments
    - Need to use recursion for all iterations
    - Pro:
      * Language can be learned faster
      * Programs can be written faster
    - Con:
      * Efficiency – slower execution
  + Symbolic Expressions (S\_Expressions)
    - Main data structure
    - Lisp programs themselves are written in S\_Expressions
    - BNF:
      * <S\_Expression> 🡪 <atom> | “(“ <S\_Expression> “.” <S\_Expression> “)” {There must be at least one blank before/after the period}
      * <atom> 🡪 any string of printable characters except for blanks, “ ( “, “ ) ”, “ ’ ”
    - Conservative (Cons) Expression:
      * “(“ <S\_Expression> “.” <S\_Expression> “)”
        + The first <S\_Expression> is known as the car part
        + The second <S\_Expression> is known as the cdr part
    - Special atom “()”, called nil, denotes the empty S\_Expression
    - <atom> includes numbers (like integers, floats, etc)
      * (CS316 . ABC)
      * (CS?#B . 123)
      * (CS316 . (B#5.7 . 58E-7) )
      * Can be nested to any depth
        + ( (2 . 5 ) . ( ( ABC . CS316) . ( B# . 58E07 ) ) )
    - Cons Expressions are implemented in the heap
    - Implementation-wise, Cons Expressions are binary trees
  + The S\_Expressions of the form (e1 . (e2 . … (en . () ) … ) ) are regarded as lists of *n* elements e1, e2, … , en [and abbreviated as (e1 e2 en), and there’s at least one blank to separate ei ]
    - Each ei may be any S\_Expression/list nested to any depth
  + Example:
    - ( 1 2 3 4) = ( 1 . ( 2 . ( 3 . ( 4 . ( ) ) ) ) )
    - ( 1 ( A B ) 3 ( C D ) ) = ( 1 . ( ( A . ( B . ( ) ) ) . ( 3 . ( ( C . ( D . ( ) ) ) . ( ) ) ) ) )
      * (AB) = ( A . ( B . ( ) ) )
      * (CD) = ( C . ( D . ( ) ) )
    - All lisp operator/function applications take the form ( f e1 e2 … en )
      * Where f = operator/function
      * e1 e2 … en = arguments
  + Example: (f ( a 1 2 3 ) ) [f and a are functions]
    - ( f ‘(a 1 2 3) ) – applying f to the list of 4 elements; a, 1, 2, 3
      * [f is a function, a is an atom]
    - Apply ‘ to S\_Expressions/lists when they are used as data arguments to functions
    - (car ‘(e1 . e2)) 🡺 e1
    - (cdr ‘(e1 . e2)) 🡺 e2
    - (cons ‘e1 ‘e2) 🡺 (e1 . e2)
  + (car ‘((1 2) . (3 4) ) ) 🡺 (1 2)
  + (cdr ‘((1 2) . (3 4) ) ) 🡺 (3 4)
  + (cons ‘(1 2) ‘(3 4) ) 🡺 ( (1 2) . (3 4) )
  + (car ‘(e1 e2 … en)) 🡺 e1 [the head of the list]
  + (cdr ‘(e1 e1 … en)) 🡺 (e2 … en) [the tail list]
  + (cons ‘e1 ‘(e2 … en)) 🡺 (e1 e2 … en)