Lecture 7
Inductive Sets of Data & Recursive Procedures

T. METIN SEZGIN

Lecture Nuggets

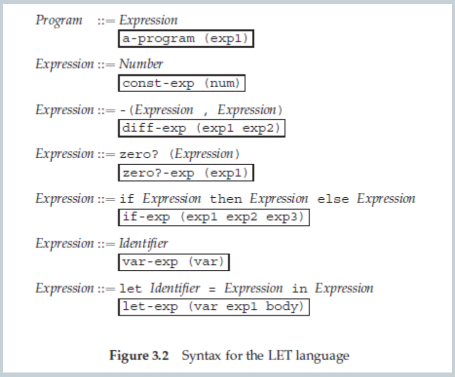
- Recursion is important
- We can specify data recursively
  - Inductive data specification
  - Defining sets using grammars
  - Induction
- We can use prove properties of recursively defined data
- We can write programs recursively
  - Smaller sub-problem principle (wishful thinking)
  - Examples
  - Auxiliary procedures

Nugget

Recursion is important

Recursion is important

- Recursion is important
  - Syntax in programming languages is nested
- Data definitions can be recursive
- Procedure definitions can be recursive



```
Program    ::= Expression
               a-program (exp1)

Expression ::= Number
               const-exp (num)

Expression ::= - (Expression , Expression)
               diff-exp (exp1 exp2)

Expression ::= zero? (Expression)
               zero?-exp (exp1)

Expression ::= if Expression then Expression else Expression
               if-exp (exp1 exp2 exp3)

Expression ::= Identifier
               var-exp (var)

Expression ::= let Identifier = Expression in Expression
               let-exp (var exp1 body)
```

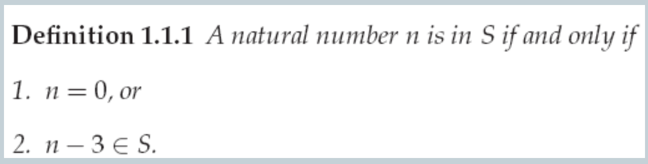
Figure 3.2 Syntax for the LET language

Nugget

We can define data recursively

Recursion example

- Inductive specification of a subset of natural numbers $N = \{0, 1, 2, \ldots\}$



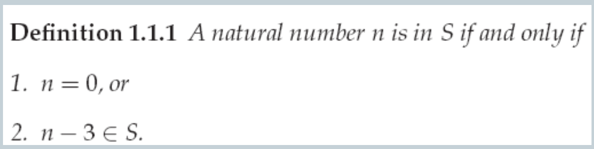
Definition 1.1.1 *A natural number n is in S if and only if*

1. $n = 0$, *or*
2. $n - 3 \in S$.

- Which subset of *N* is this?
- Is 6 in S?

Simple procedure for testing membership

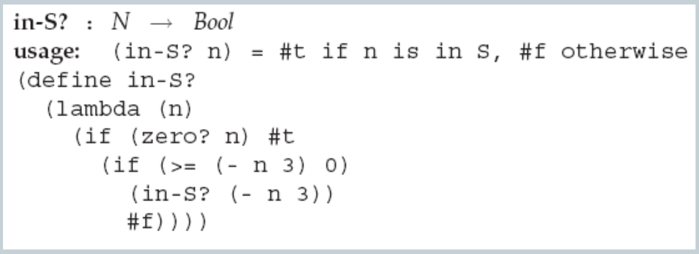
- Write a procedure that follows the definition
- Remember the definition



Definition 1.1.1 *A natural number n is in S if and only if*

1. $n = 0$, *or*
2. $n - 3 \in S$.

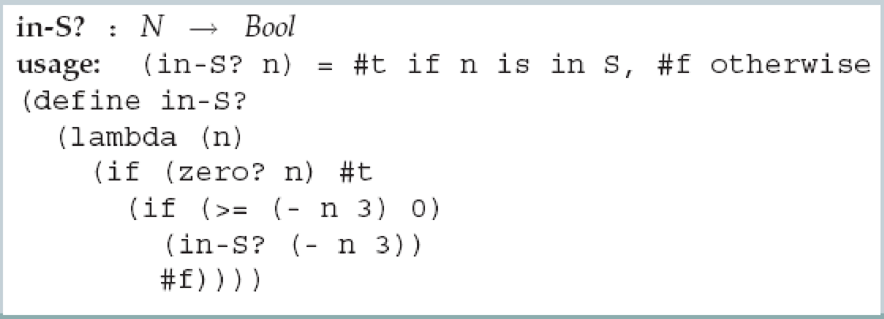
- And the procedure



```
in-S? : N → Bool
usage:  (in-S? n) = #t if n is in S, #f otherwise
(define in-S?
  (lambda (n)
    (if (zero? n) #t
      (if (>= (- n 3) 0)
        (in-S? (- n 3))
        #f))))
```

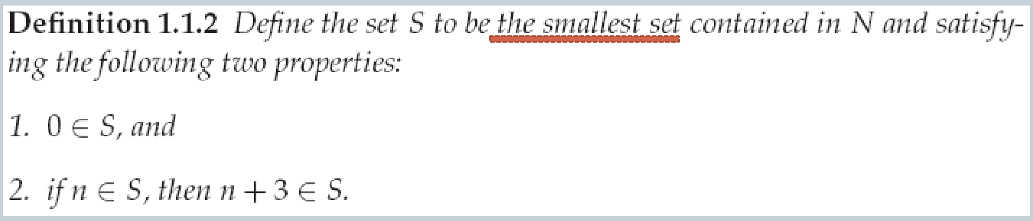
Simple procedure for testing membership

- More about the procedure
  - Contract
  - Domain
  - Co-Domain (range)
  - Usage
  - Argument



```
in-S? : N → Bool
usage:  (in-S? n) = #t if n is in S, #f otherwise
(define in-S?
  (lambda (n)
    (if (zero? n) #t
      (if (>= (- n 3) 0)
        (in-S? (- n 3))
        #f))))
```

Alternative definition of S



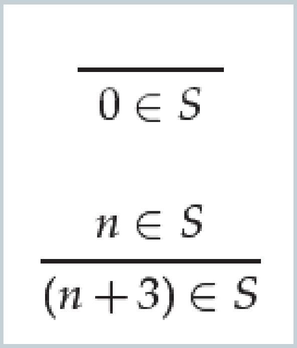
Definition 1.1.2 *Define the set S to be the smallest set contained in $\mathbb{N}$ and satisfying the following two properties:*

1. $0 \in S$, *and*
2. *if* $n \in S$, *then* $n + 3 \in S$.

- Show that “the smallest set” constraint is needed
- Show that there is only one set that is smallest

Yet another way of defining S

- Rule of Inference
- Concepts
  - Hypothesis (antecedent)
  - Conclusion (consequent)
  - Implies
  - Implicit AND
  - Axiom



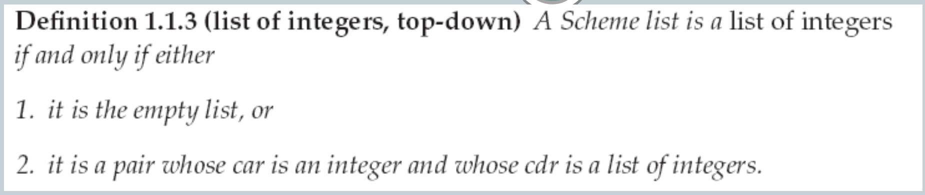
$$\frac{}{0 \in S}$$

$$\frac{n \in S}{(n+3) \in S}$$

Three different ways of defining S

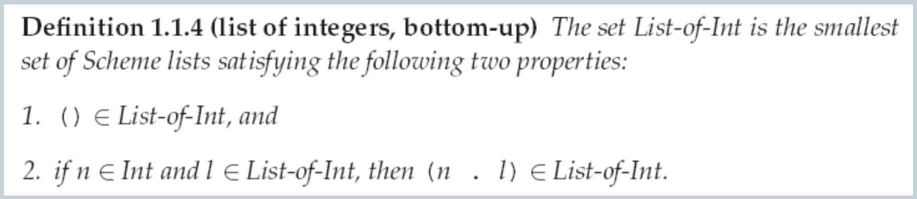
- Top-down
  - The recursion ends at the base case
- Bottom-up
  - Induction starts at the base case
- Rules-of-inference
  - Must find a sequence of derivations

Defining list of integers



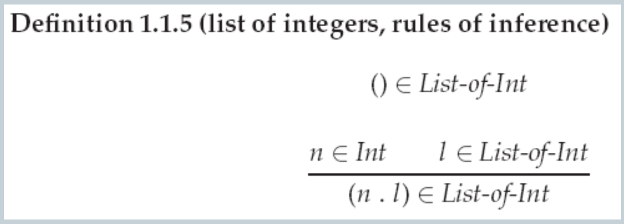
Definition 1.1.3 (list of integers, top-down) *A Scheme list is a* list of integers *if and only if either*

1. *it is the empty list, or*
2. *it is a pair whose car is an integer and whose cdr is a list of integers.*



Definition 1.1.4 (list of integers, bottom-up) *The set List-of-Int is the smallest set of Scheme lists satisfying the following two properties:*

1. $() \in$ *List-of-Int, and*
2. *if* $n \in$ *Int and* $l \in$ *List-of-Int, then* $(n \;.\; l) \in$ *List-of-Int.*



Definition 1.1.5 (list of integers, rules of inference)

$$() \in \mathit{List\text{-}of\text{-}Int}$$

$$\frac{n \in \mathit{Int} \qquad l \in \mathit{List\text{-}of\text{-}Int}}{(n \;.\; l) \in \mathit{List\text{-}of\text{-}Int}}$$

Example

- Show that (-7 3 14) is a list of integers:

`(-7 . (3 . (14 . ())))`

Example

- Show that (-7 3 14) is a list of integers:

`(-7 . (3 . (14 . ())))`

- Derivation (deduction tree)

$$\dfrac{\texttt{-7} \in N \qquad \dfrac{\texttt{3} \in N \qquad \dfrac{\texttt{14} \in N \qquad () \in \textit{List-of-Int}}{\texttt{(14 . ())} \in \textit{List-of-Int}}}{\texttt{(3 . (14 . ()))} \in \textit{List-of-Int}}}{\texttt{(-7 . (3 . (14 . ())))} \in \textit{List-of-Int}}$$

Defining Sets Using Grammars

$$\mathit{List\text{-}of\text{-}Int} ::= (\,)$$
$$\mathit{List\text{-}of\text{-}Int} ::= (\mathit{Int} \ . \ \mathit{List\text{-}of\text{-}Int})$$

- Components of a grammar
  - Terminals
  - Non-terminals (syntactic categories)
  - Productions (no context)
  - Optional bits
  - Naming conventions $e \in \mathit{Expression}$
- BNF, CNF
- Kleene notation
  - Star {<exp>}*, Plus {<exp>}$^{+}$, Separated list Plus {<exp>}$^{+(,)}$

Grammar example

- S-lists

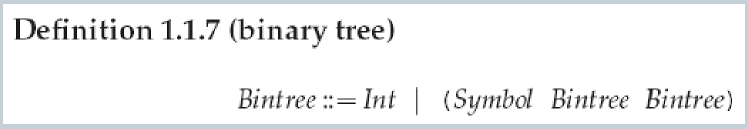
Definition 1.1.6 (s-list, s-exp)

$$\begin{aligned} \mathit{S\text{-}list} &::= (\{\mathit{S\text{-}exp}\}^{*}) \\ \mathit{S\text{-}exp} &::= \mathit{Symbol} \mid \mathit{S\text{-}list} \end{aligned}$$

- Examples
- S-list -> ()
- S-exp -> x
- S-list -> (x)
- S-exp -> (x)
- S-list -> ((x) x (x) ((x) x (x)))

Grammar example

- Binary Trees



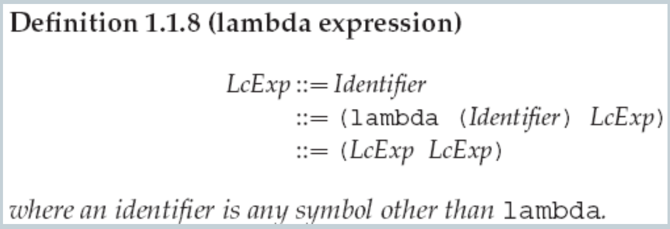
Definition 1.1.7 (binary tree)

$$Bintree ::= Int \mid (Symbol\ \ Bintree\ \ Bintree)$$

- Examples

Grammar example

- Lambda Calculus



Definition 1.1.8 (lambda expression)

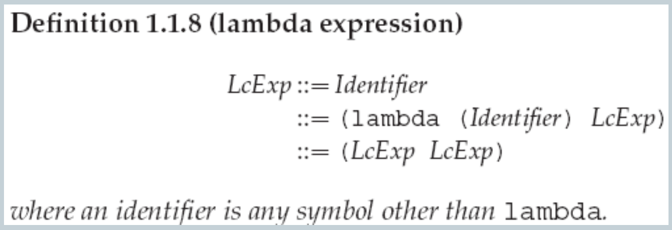
$$\begin{aligned} LcExp &::= Identifier \\ &::= (\texttt{lambda}\ \ (Identifier)\ \ LcExp) \\ &::= (LcExp\ \ LcExp) \end{aligned}$$

where an identifier is any symbol other than `lambda`.

- Examples
- (lambda (x) x)
- (lambda (x) (lambda (y) z))

Grammar example

- Lambda Calculus



Definition 1.1.8 (lambda expression)

$$
\begin{aligned}
LcExp &::= Identifier \\
&::= (\texttt{lambda}\ (Identifier)\ LcExp) \\
&::= (LcExp\ \ LcExp)
\end{aligned}
$$

where an identifier is any symbol other than `lambda`.

- Concepts
  - Variables
  - Bound variable

Nugget

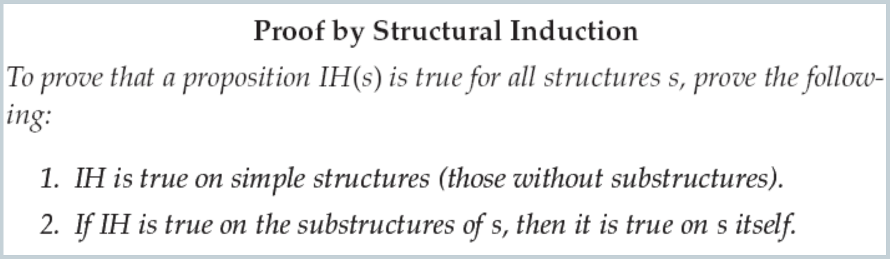
We can use prove properties of recursively defined data

Induction

- A method for formal proofs
- Steps
  - Define an induction hypothesis IH: Int □ bool
  - Prove base case IH(0)
  - Prove that IH(k) □ IH(k+1)
    - or more generally IH(k') for k'<=k □ IH(k+1)

Structural Induction

- A method for formal proofs
- Steps
  - Define an induction hypothesis IH: Int □ bool
  - Prove base case IH(0)
  - Prove that IH(k) □ IH(k+1)
    - or more generally IH(k') for k'<=k □ IH(k+1)



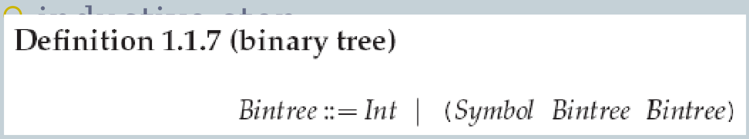
Proof by Structural Induction

To prove that a proposition IH(s) is true for all structures s, prove the following:

1. *IH is true on simple structures (those without substructures).*
2. *If IH is true on the substructures of s, then it is true on s itself.*

Induction Example

- Prove that binary trees have odd number of nodes
  - Use structural induction
- Define IH(k)
  - Any tree of size k has odd number of elements
- Prove
  - base case



Definition 1.1.7 (binary tree)

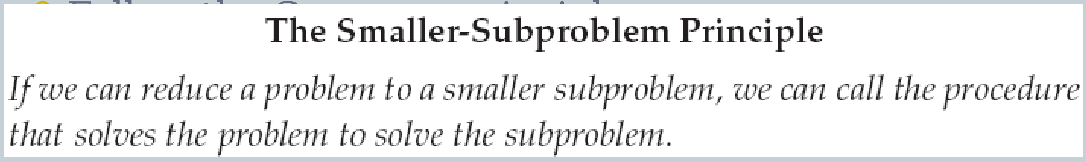
$$Bintree ::= Int \mid (Symbol\ \ Bintree\ \ Bintree)$$

Nugget

We can solve problems using recursion

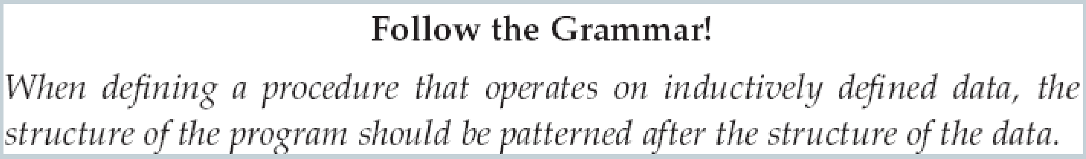
Deriving Recursive Programs

- Recursive programs are easy to write if you follow two principles
  - Smaller-sub-problem principle (aka divide and conquer).
  - Follow the Grammar principle



The Smaller-Subproblem Principle

If we can reduce a problem to a smaller subproblem, we can call the procedure that solves the problem to solve the subproblem.

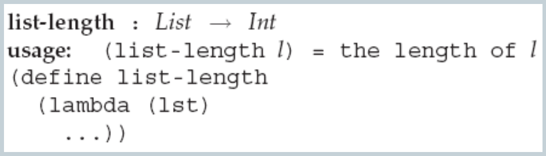


Follow the Grammar!

When defining a procedure that operates on inductively defined data, the structure of the program should be patterned after the structure of the data.

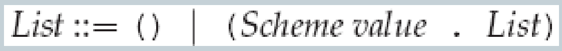
Recursive Procedure Example

- Write a new function list-length
- Everyone should be able to go this far

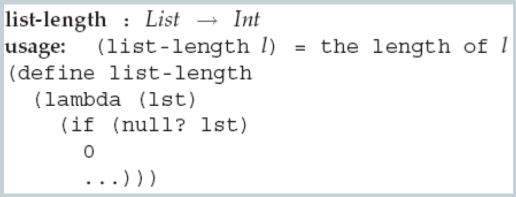


```
list-length : List → Int
usage:  (list-length l) = the length of l
(define list-length
  (lambda (lst)
    ...))
```

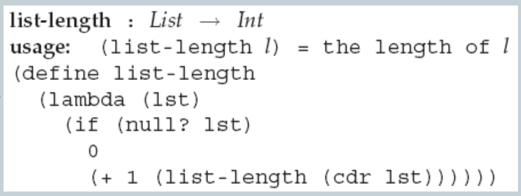
- Let the definition of **`list`** guide you



$List ::= () \mid (Scheme\ value\ .\ List)$



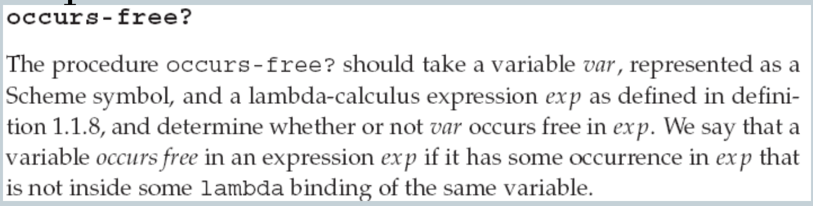
```
list-length : List → Int
usage:  (list-length l) = the length of l
(define list-length
  (lambda (lst)
    (if (null? lst)
      0
      ...)))
```



```
list-length : List → Int
usage:  (list-length l) = the length of l
(define list-length
  (lambda (lst)
    (if (null? lst)
      0
      (+ 1 (list-length (cdr lst))))))
```

Another Example

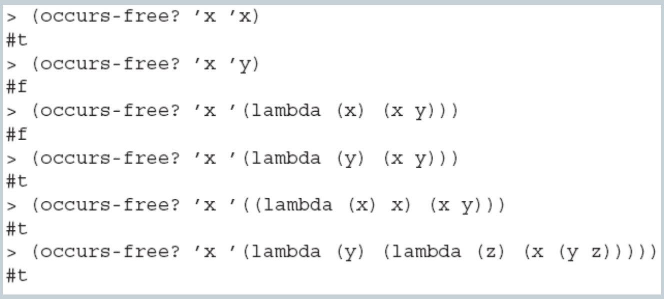
- Implement occurs-free?



occurs-free?

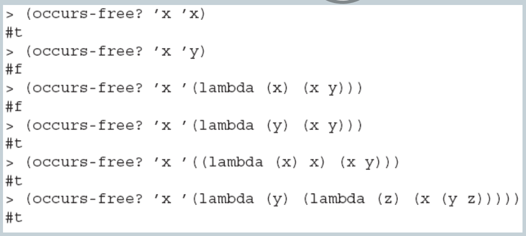
The procedure `occurs-free?` should take a variable *var*, represented as a Scheme symbol, and a lambda-calculus expression *exp* as defined in definition 1.1.8, and determine whether or not *var* occurs free in *exp*. We say that a variable *occurs free* in an expression *exp* if it has some occurrence in *exp* that is not inside some `lambda` binding of the same variable.

- Such that

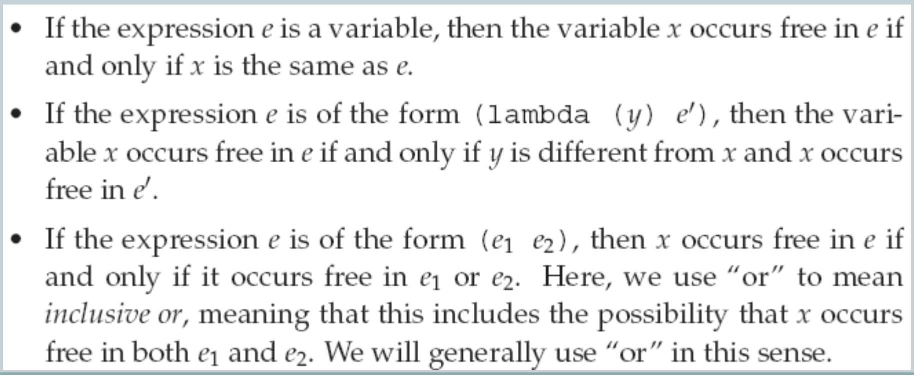


```
> (occurs-free? 'x 'x)
#t
> (occurs-free? 'x 'y)
#f
> (occurs-free? 'x '(lambda (x) (x y)))
#f
> (occurs-free? 'x '(lambda (y) (x y)))
#t
> (occurs-free? 'x '((lambda (x) x) (x y)))
#t
> (occurs-free? 'x '(lambda (y) (lambda (z) (x (y z)))))
#t
```

The rules of occurs-free?

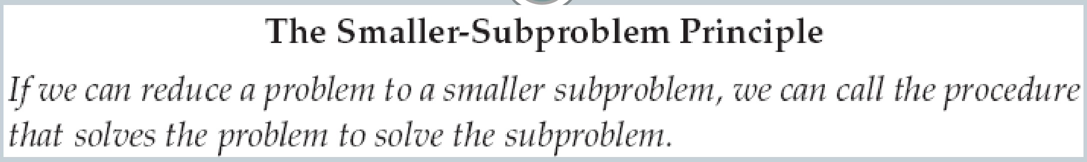


```
> (occurs-free? 'x 'x)
#t
> (occurs-free? 'x 'y)
#f
> (occurs-free? 'x '(lambda (x) (x y)))
#f
> (occurs-free? 'x '(lambda (y) (x y)))
#t
> (occurs-free? 'x '((lambda (x) x) (x y)))
#t
> (occurs-free? 'x '(lambda (y) (lambda (z) (x (y z)))))
#t
```



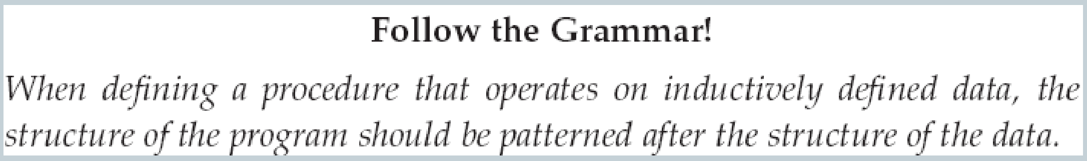
- If the expression e is a variable, then the variable x occurs free in e if and only if x is the same as e.
- If the expression e is of the form `(lambda (`y`)` e'`)`, then the variable x occurs free in e if and only if y is different from x and x occurs free in e'.
- If the expression e is of the form `(`e_1 e_2`)`, then x occurs free in e if and only if it occurs free in e_1 or e_2. Here, we use "or" to mean *inclusive or*, meaning that this includes the possibility that x occurs free in both e_1 and e_2. We will generally use "or" in this sense.

How do we go about the implementation?



The Smaller-Subproblem Principle

If we can reduce a problem to a smaller subproblem, we can call the procedure that solves the problem to solve the subproblem.

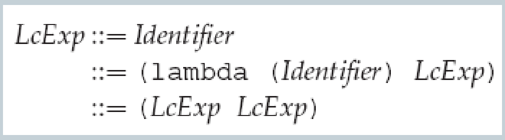


Follow the Grammar!

When defining a procedure that operates on inductively defined data, the structure of the program should be patterned after the structure of the data.

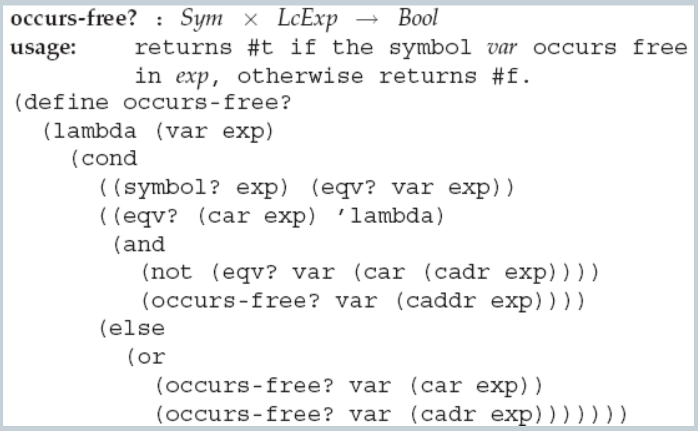
How do we go about the implementation?

- The grammar



```
LcExp ::= Identifier
      ::= (lambda (Identifier) LcExp)
      ::= (LcExp LcExp)
```

- The procedure



occurs-free? : *Sym* × *LcExp* → *Bool*
usage: returns #t if the symbol *var* occurs free in *exp*, otherwise returns #f.

```
(define occurs-free?
  (lambda (var exp)
    (cond
      ((symbol? exp) (eqv? var exp))
      ((eqv? (car exp) 'lambda)
       (and
         (not (eqv? var (car (cadr exp))))
         (occurs-free? var (caddr exp))))
      (else
        (or
          (occurs-free? var (car exp))
          (occurs-free? var (cadr exp)))))))
```