

Lecture 6

Inductive Sets of Data & Recursive Procedures



T. METIN SEZGIN

Lecture 05 – Review Lists and recursion



T. METIN SEZGIN

Lecture notes



Higher Order Procedures:

y functions take other functions as input. for ex: $f \circ g(x)$

It is "higher order procedure" because it takes "procedures" as input.

Nugget: Higher Order Procedures

Examples: $1+2+\dots+100 = (100 \cdot 101) / 2$
 $1+4+\dots+100^2 = (100 \cdot 101 \cdot 201) / 6$
 $1+1/3^2+\dots+1/101^2 = \pi^2/8$

$$\sum_{k=1}^{100} k$$
$$\sum_{k=1}^{100} k^2$$

$$\sum_{k=1, \text{ odd}}^{101} k^{-2}$$

Function which is General-Sum for all series:

(define (sum term a next b)

(if (> a b)

0

(+ (term a)

(sum term (next a) (next b)))))

- Hasan Buhurcu
- Arda Adalar
- Ali Avcı

Lecture notes

Consing Up A list

in a recursive algorithm usually this case happens:


Cons $\underbrace{(\dots)}_{1^{st}}$ $\underbrace{(\dots)}_{2^{nd}}$ and the second part include the recursion the logic is:

 ? → this will wait for recursion

some kind of data

at the 2nd step of algorithm:
again first part will be determined
and it will join the list

it means data
1st step:  ?

2nd step:  ?

⋮

at the end of
the algorithm

there will be a base
case usually it will
return '1'

what does it mean
it means at the end
there will not be a
next element



Cdr'ing down :

It means repeatedly applying the cdr operation to a list until
reaching the end of the list or a specified element in the list.

Lecture notes



```
(define (sum-integers a b)
  (if (> a b)
```

0

```
    (+ a (sum-integers
          (+ 1 a) b))))
```

```
(define (sum-squares a b)
  (if (> a b)
```

0

```
    (+ (square a) (sum-squares
                  (+ 1 a) b))))
```

```
(define (pi-sum a b)
  (if (> a b)
```

0

```
    (+ (/ 1 (square a))
      (pi-sum (+ a 2) b))))
```

identity func.

```
(define (sum term a next b)
```

```
  (if (> a b)
```

0

```
    (+ (term a)
      (sum term (next a)
            next b))))
```

continue summing
from the next element.

$(\underbrace{\text{number} \rightarrow \text{number}}_{\text{term}}, \underbrace{\text{number}}_a, \underbrace{\text{number} \rightarrow \text{number}}_{\text{next}}, \underbrace{\text{number}}_b) \rightarrow \text{number}$

sum.

Lecture notes



mapping:

(define (map proc lst) → define map func. takes 2 args : procedure
(if null? lst) → if null, return nil. list

nil
(cons (proc (car lst)) → if not, apply procedure to first element, then
(map proc (cdr lst)))) return the rest of the list.

★ mapping is a process of applying a proc. to each element of a list

Lecture notes



filtering:

```
(define (filter pred lst)
  (cond ((null? lst) nil) ←
        ((pred (car lst)) ← not empty, applying predicate to the first elt.
         (cons (car lst) (filter pred (cdr lst))))) ← cons. a new list w/
        (else (filter pred (cdr lst))))) ← the rest of the
                                           elts.
```

★ filtering is a process of selecting a subset of items from a collection based on a specific criterion (predicate)

accumulating:

```
(define reduce op init lst)
  (if (null? lst)
      init
      (op (car lst)
          (reduce op init (cdr lst)))))
```

Diagram illustrating the recursive structure of the `reduce` function:

- A dashed line connects the `op` parameter and the `(reduce op init (cdr lst))` recursive call.
- An arrow points from the `op` parameter to the text "applying op here".
- An arrow points from the recursive call to the text "recursively calling here".

★ accumulating is a process of applying the operation to a set of inputs.

Lecture 6

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 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

```

Identifier:
    IDENTIFIER

QualifiedIdentifier:
    Identifier { . Identifier }

QualifiedIdentifierList:
    QualifiedIdentifier { , QualifiedIdentifier }

CompilationUnit:
    [[Annotations] package QualifiedIdentifier ;]
                                {ImportDeclaration} {TypeDeclaration}

ImportDeclaration:
    import [static] Identifier { . Identifier } [ . * ] ;

TypeDeclaration:
    ClassOrInterfaceDeclaration
    ;

ClassOrInterfaceDeclaration:
    {Modifier} (ClassDeclaration | InterfaceDeclaration)

ClassDeclaration:
    NormalClassDeclaration
    EnumDeclaration

InterfaceDeclaration:
    NormalInterfaceDeclaration
    AnnotationTypeDeclaration

NormalClassDeclaration:
    class Identifier [TypeParameters]
                                [extends Type] [implements TypeList] ClassBody

EnumDeclaration:
    enum Identifier [implements TypeList] EnumBody

NormalInterfaceDeclaration:
    interface Identifier [TypeParameters] [extends TypeList] InterfaceBody

AnnotationTypeDeclaration:
    @ interface Identifier AnnotationTypeBody

```

```

ClassBody:
    { { ClassBodyDeclaration } }

ClassBodyDeclaration:
    ;
    {Modifier} MemberDecl
    [static] Block

MemberDecl:
    MethodOrFieldDecl
    void Identifier VoidMethodDeclaratorRest
    Identifier ConstructorDeclaratorRest
    GenericMethodOrConstructorDecl
    ClassDeclaration
    InterfaceDeclaration

MethodOrFieldDecl:
    Type Identifier MethodOrFieldRest

MethodOrFieldRest:
    FieldDeclaratorsRest ;
    MethodDeclaratorRest

FieldDeclaratorsRest:
    VariableDeclaratorRest { , VariableDeclarator }

MethodDeclaratorRest:
    FormalParameters {[}] [throws QualifiedIdentifierList] (Block | ;)

VoidMethodDeclaratorRest:
    FormalParameters [throws QualifiedIdentifierList] (Block | ;)

ConstructorDeclaratorRest:
    FormalParameters [throws QualifiedIdentifierList] Block

GenericMethodOrConstructorDecl:
    TypeParameters GenericMethodOrConstructorRest

GenericMethodOrConstructorRest:
    (Type | void) Identifier MethodDeclaratorRest
    Identifier ConstructorDeclaratorRest

```

Nugget



We can define data recursively

Recursion example



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

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? : $N \rightarrow Bool$`

`usage: (in-S? n) = #t if n is in S, #f otherwise`

`(define in-S?`

`(lambda (n)`

`(if (zero? n) #t`

`(if (\geq (- 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? :  $N \rightarrow Bool$   
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 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

$$\frac{\overline{0 \in S}}{n \in S} \frac{n \in S}{\overline{(n + 3) \in S}}$$

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. *$() \in \text{List-of-Int}$, and*
2. *if $n \in \text{Int}$ and $l \in \text{List-of-Int}$, then $(n . l) \in \text{List-of-Int}$.*

Definition 1.1.5 (list of integers, rules of inference)

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

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

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)

$$\frac{-7 \in N \quad \frac{3 \in N \quad \frac{14 \in N \quad () \in List-of-Int}{(14 \ . \ ()) \in List-of-Int}}{(3 \ . \ (14 \ . \ ())) \in List-of-Int}}{(-7 \ . \ (3 \ . \ (14 \ . \ ()))) \in List-of-Int}$$

Defining Sets Using Grammars



List-of-Int ::= ()

List-of-Int ::= (Int . List-of-Int)

- Components of a grammar

- Terminals
- Non-terminals (syntactic categories)
- Productions (no context)
- Optional bits
- Naming conventions $e \in Expression$

- BNF, CNF

- Kleene notation

- Star $\{<exp>\}^*$, Plus $\{<exp>\}^+$, Separated list Plus $\{<exp>\}^{+,()}$

Grammar example



- S-lists

Definition 1.1.6 (s-list, s-exp)

$$S\text{-list} ::= (\{S\text{-exp}\}^*)$$
$$S\text{-exp} ::= \text{Symbol} \mid S\text{-list}$$

- Examples
- $S\text{-list} \rightarrow ()$
- $S\text{-exp} \rightarrow x$
- $S\text{-list} \rightarrow (x)$
- $S\text{-exp} \rightarrow (x)$
- $S\text{-list} \rightarrow ((x) x (x) ((x) x (x)))$

Grammar example



- Binary Trees

Definition 1.1.7 (binary tree)

$$\textit{Bintree} ::= \textit{Int} \mid (\textit{Symbol} \textit{Bintree} \textit{Bintree})$$

- Examples

Grammar example



- Lambda Calculus

Definition 1.1.8 (lambda expression)

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

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)

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

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 $IH: \text{Int} \rightarrow \text{bool}$
 - Prove base case $IH(0)$
 - Prove that $IH(k) \rightarrow IH(k+1)$
 - ✦ or more generally $IH(k') \text{ for } k' \leq k \rightarrow IH(k+1)$

Structural Induction



- A method for formal proofs
- Steps
 - Define an induction hypothesis $IH: \text{Int} \rightarrow \text{bool}$
 - Prove base case $IH(o)$
 - Prove that $IH(k) \rightarrow IH(k+1)$
 - ✦ or more generally $IH(k') \text{ for } k' \leq k \rightarrow IH(k+1)$

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
 - inductive step

Definition 1.1.7 (binary tree)

$$\text{Bintree} ::= \text{Int} \mid (\text{Symbol Bintree Bintree})$$

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 ::= () | (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 $(\text{lambda } (y) e')$, 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)))))))
```