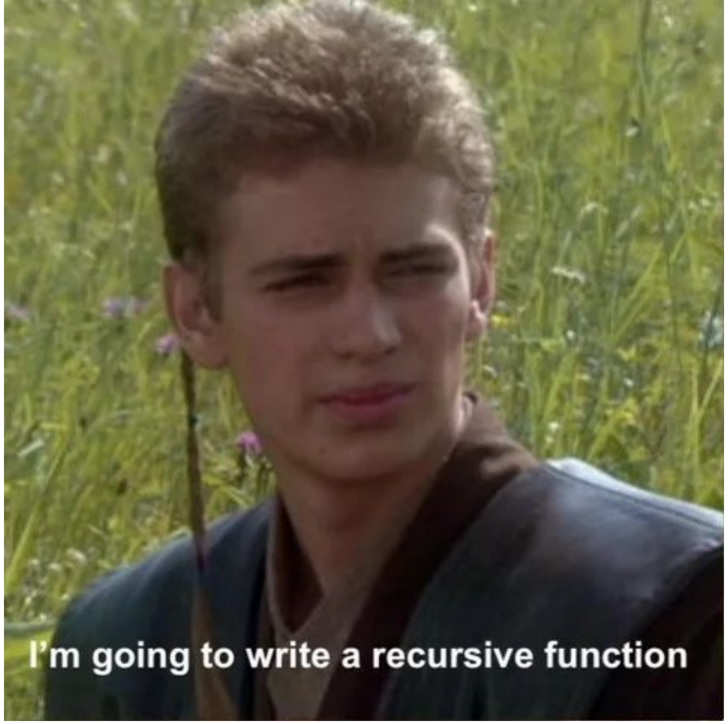
Lecture 6 – Review
Inductive Sets of Data &
Recursive Procedures

T. METIN SEZGIN

Nugget

Recursion is important

You need to get it right


I’m going to write a recursive function


I'm going to write a recursive function
With a base case, right?


I'm going to write a recursive function
With a base case, right?


I'm going to write a recursive function
With a base case, right?
I'm going to write a recursive function


I'm going to write a recursive function
With a base case, right?
I'm going to write a recursive function
With a base case, right?


I'm going to write a recursive function
With a base case, right?
I'm going to write a recursive function
With a base case, right?

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

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. $() \in \textit{List-of-Int}$, *and*

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

Definition 1.1.5 (list of integers, rules of inference)

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

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

Grammar example

- Lambda Calculus

Definition 1.1.8 (lambda expression)

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

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

- Concepts
  - Variables
  - Bound variable

Nugget

We can use prove properties of recursively defined data

Lecture 7
Recursive Procedures

T. METIN SEZGIN

Lecture Nuggets

- We can write programs recursively
  - We can apply the smaller sub-problem principle (wishful thinking)
  - Examples
- If needed we can make use of Auxiliary procedures
- Sometimes it is easier to write more general procedures

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* $\longrightarrow$ *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* $\longrightarrow$ *Int*
usage: (list-length *l*) = the length of *l*

```
(define list-length
  (lambda (lst)
    (if (null? lst)
      0
      ...)))
```

list-length : *List* $\longrightarrow$ *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

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

- The procedure

occurs-free? : $Sym \times LcExp \rightarrow Bool$

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)))))))
```

Nugget

If needed, we can use auxiliary procedures

subst

subst

The procedure `subst` should take three arguments: two symbols, `new` and `old`, and an s-list, `slist`. All elements of `slist` are examined, and a new list is returned that is similar to `slist` but with all occurrences of `old` replaced by instances of `new`.

```
> (subst 'a 'b '((b c) (b () d)))
((a c) (a () d))
```

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

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

$$
\begin{aligned}
\textit{S-list} &::= () \\
&::= (\textit{S-exp} \;\; . \;\; \textit{S-list}) \\
\textit{S-exp} &::= \textit{Symbol} \mid \textit{S-list}
\end{aligned}
$$

- The procedure

subst : $\textit{Sym} \times \textit{Sym} \times \textit{S-list} \to \textit{S-list}$

```
(define subst
  (lambda (new old slist)
    ...))
```

subst-in-s-exp : $\textit{Sym} \times \textit{Sym} \times \textit{S-exp} \to \textit{S-exp}$

```
(define subst-in-s-exp
  (lambda (new old sexp)
    ...))
```

How do we go about the implementation?

- The grammar

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

$$\textit{S-list} ::= ()$$
$$::= (\textit{S-exp} \;.\; \textit{S-list})$$
$$\textit{S-exp} ::= \textit{Symbol} \mid \textit{S-list}$$

- The procedure

subst : $\textit{Sym} \times \textit{Sym} \times \textit{S-list} \to \textit{S-list}$

```
(define subst
  (lambda (new old slist)
    (if (null? slist)
      '()
      ...)))
```

How do we go about the implementation?

- The grammar

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

$$\begin{aligned} S\text{-}list &::= () \\ &::= (S\text{-}exp \; . \; S\text{-}list) \\ S\text{-}exp &::= Symbol \mid S\text{-}list \end{aligned}$$

- The procedure

subst : $Sym \times Sym \times S\text{-}list \rightarrow S\text{-}list$

```
(define subst
  (lambda (new old slist)
    (if (null? slist)
      '()
      (cons
        (subst-in-s-exp new old (car slist))
        (subst new old (cdr slist))))))
```

How do we go about the implementation?

- The grammar

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

$$\begin{aligned}\textit{S-list} &::= () \\ &::= (\textit{S-exp} \;.\; \textit{S-list}) \\ \textit{S-exp} &::= \textit{Symbol} \mid \textit{S-list}\end{aligned}$$

- The procedure

subst : $\textit{Sym} \times \textit{Sym} \times \textit{S-list} \rightarrow \textit{S-list}$

```
(define subst
  (lambda (new old slist)
    (if (null? slist)
      '()
      (cons
        (subst-in-s-exp new old (car slist))
        (subst new old (cdr slist))))))
```

subst-in-s-exp : $\textit{Sym} \times \textit{Sym} \times \textit{S-exp} \rightarrow \textit{S-exp}$

```
(define subst-in-s-exp
  (lambda (new old sexp)
    (if (symbol? sexp)
      (if (eqv? sexp old) new sexp)
      (subst new old sexp))))
```

Things to note

- The procedures are mutually recursive
- The trick of decomposing procedures for each syntactic type is important
  - Simplifies our design

$$
\begin{aligned}
\textit{S-list} &::= () \\
&::= (\textit{S-exp} \;\; . \;\; \textit{S-list}) \\
\textit{S-exp} &::= \textit{Symbol} \;\mid\; \textit{S-list}
\end{aligned}
$$

subst : $\textit{Sym} \times \textit{Sym} \times \textit{S-list} \to \textit{S-list}$

```
(define subst
  (lambda (new old slist)
    (if (null? slist)
      '()
      (cons
        (subst-in-s-exp new old (car slist))
        (subst new old (cdr slist))))))
```

subst-in-s-exp : $\textit{Sym} \times \textit{Sym} \times \textit{S-exp} \to \textit{S-exp}$

```
(define subst-in-s-exp
  (lambda (new old sexp)
    (if (symbol? sexp)
      (if (eqv? sexp old) new sexp)
      (subst new old sexp))))
```

Take home message

Follow the Grammar

More precisely:

- Write one procedure for each nonterminal in the grammar. The procedure will be responsible for handling the data corresponding to that nonterminal, and nothing else.
- In each procedure, write one alternative for each production corresponding to that nonterminal. You may need additional case structure, but this will get you started. For each nonterminal that appears in the right-hand side, write a recursive call to the procedure for that nonterminal.

Nugget

Sometimes it is easier to write more general procedures

A more complex example

- Consider the procedure **`number-elements`**
- This procedure should take a list $(\mathbf{v}_0\ \mathbf{v}_1\ \mathbf{v}_2\ \ldots)$ and return $((0\ \mathbf{v}_0)\ (1\ \mathbf{v}_1)\ \ldots))$.
- Remember the grammar

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

- The problem
  - No obvious way to build **`(number-elements lst)`** from **`(number-elements (cdr lst))`**
- The solution
  - Implement something ***more general***
  - Implement **`number-elements-from`**

$$\textbf{number-elements-from} : \mathit{Listof}(\mathit{SchemeVal}) \times \mathit{Int} \to \mathit{Listof}(\mathit{List}(\mathit{Int}, \mathit{SchemeVal}))$$

number-elements-from

number-elements-from : $Listof(SchemeVal) \times Int \to Listof(List(Int, SchemeVal))$

usage: (number-elements-from '(v_0 v_1 v_2 ...) n)
= ((n v_0) ($n+1$ v_1) ($n+2$ v_2) ...)

```
(define number-elements-from
  (lambda (lst n)
    (if (null? lst) '()
      (cons
        (list n (car lst))
        (number-elements-from (cdr lst) (+ n 1))))))
```

number-elements : $List \to Listof(List(Int, SchemeVal))$

```
(define number-elements
  (lambda (lst)
    (number-elements-from lst 0)))
```

- How are the arguments different?
- What purpose do they serve?
  - Input list
  - Context argument (inherited attribute)

The take home message

Follow the grammar

When following the grammar doesn't help…

Generalize

Another example

- Consider **`list-sum`**

list-sum : *Listof(Int)* $\rightarrow$ *Int*

```
(define list-sum
  (lambda (loi)
    (if (null? loi)
      0
      (+ (car loi)
         (list-sum (cdr loi))))))
```

- How about vector sum?
- You can’t take cdr of vectors!

How do we go about the implementation?

Follow the grammar

When following the grammar doesn’t help…

Generalize

vector-sum

partial-vector-sum : $Vectorof(Int) \times Int \to Int$
usage: if $0 \leq n < length(v)$, then

$$\texttt{(partial-vector-sum } v\ n\texttt{)} = \sum_{i=0}^{i=n} v_i$$

```
(define partial-vector-sum
  (lambda (v n)
    (if (zero? n)
      (vector-ref v 0)
      (+ (vector-ref v n)
         (partial-vector-sum v (- n 1))))))
```

vector-sum : $Vectorof(Int) \to Int$

usage: $\texttt{(vector-sum } v\texttt{)} = \sum_{i=0}^{i=length(v)-1} v_i$

```
(define vector-sum
  (lambda (v)
    (let ((n (vector-length v)))
      (if (zero? n)
        0
        (partial-vector-sum v (- n 1))))))
```

Exercises

- EOPL Exercises
  - 1.1, 1.4, 1.6, 1.12, 1.21, 1.26, 1.34, 1.36