

Recursion on Inductive Data Types

Deriving Recursion Patterns From
Inductive Definitions

Lists as an Inductive Data Type

- **List** =
 - | **nil**
 - | **cons** Element **List** } *Type definition*
-
- nil**
(cons head tail) } *Construction forms*
- (null nil)** \rightarrow t
(null (cons head tail)) \rightarrow **nil**
(first (cons head tail)) \rightarrow *head*
(rest (cons head tail)) \rightarrow *tail* } *Elimination equations*

Induction and Recursion

- *Induction* and *recursion* are opposite perspectives of the same dual phenomenon.
- For each inductively defined data type there is an inferred pattern of recursion.
 - » The (mathematical) theory underlying this approach is the ***Martin-Löf Constructive Type Theory***. Its main applications lie in *automated programming*, *automated verification*, *type-system design*.
 - » Strong-typed functional programming languages: *Haskell* (www.haskell.org), *Standard ML* (<http://cm.bell-labs.com/cm/cs/what/smlnj/>)

Induction and Recursion

- *Inductive definition (bottom-up)*
 - defining type **List**
 - **nil** is a **List** (base case)
 - if *head* is an **Element** and *tail* is a **List**, then (**cons** *head* *tail*) is a **List** (induction case)
- *Recursive pattern (top-down)*
 - define **process** lst^{List}
 - handle *lst* is **nil** (base case)
 - handle *lst* is (**cons** *head* $tail^{\text{List}}$)
 - [if necessary] call **process** on *tail* (recursive case)

List =

| nil

| cons Element List

Recursive Pattern for Lists

```
(defun function-name (lst)  
  (cond  
    ((null lst) ...base-case-processing...)  
    (otherwise ...induction-case-processing...  
      if needed, call (function-name (rest lst))  
    )  
  )  
)
```

List =

| nil

| cons Element List

Recursive Pattern for Lists (2)

```
(defun function-name (list)
```

```
  (if list
```

one
expression



...induction-case-processing...

if needed, call (**function-name** (*rest list*))

one
expression
(optional)



...base-case-processing...

```
)
```

```
)
```

Sum

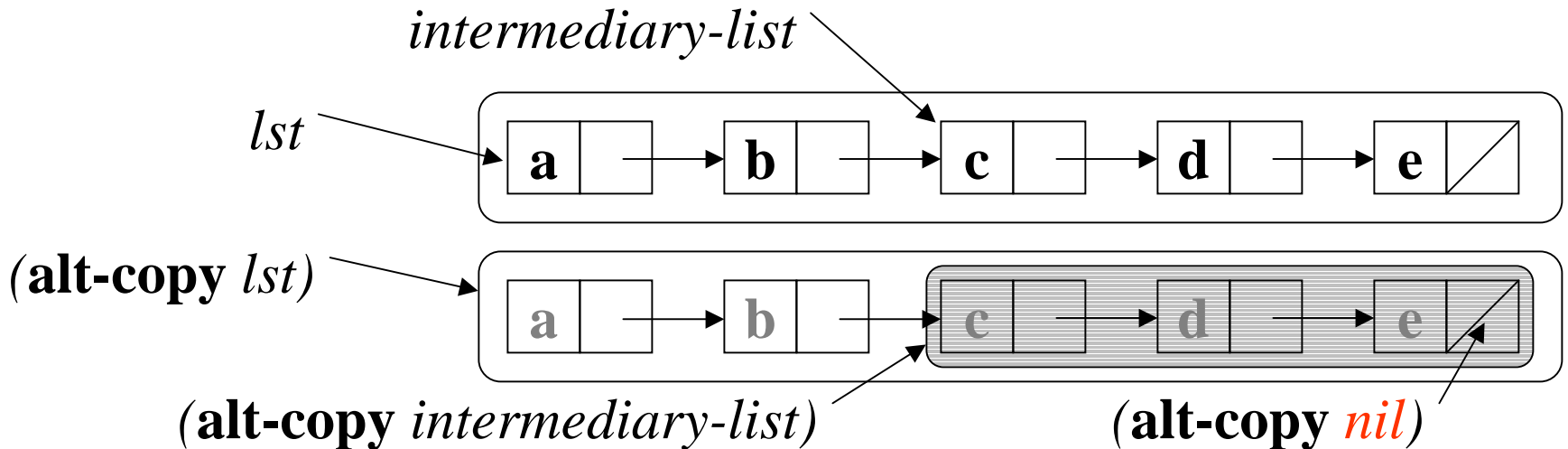
```
(defun alt-sum (lst)  
  (cond  
    ((null lst) 0)  
    (otherwise (+ (first lst) (alt-sum (rest lst))))  
  )  
)  
)
```

Copy

```
(defun alt-copy (lst)  
  (if lst  
      (cons (first lst) (alt-copy (rest lst)))  
      )  
  )
```

Copy

```
(defun alt-copy (lst)
  (if lst
      (cons (first lst) (alt-copy (rest lst)))
      )
  )
```



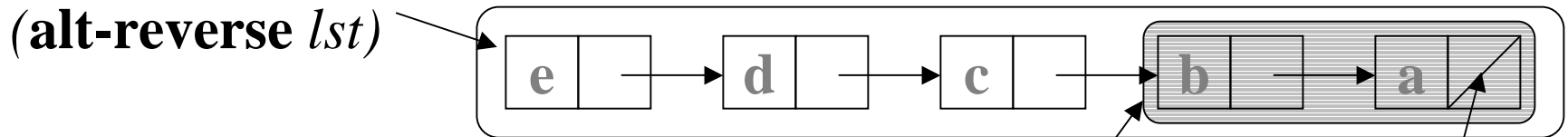
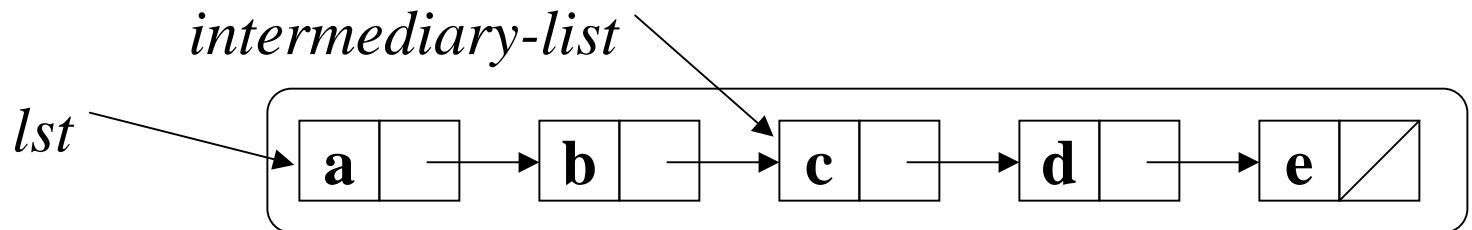
Reverse

```
(defun alt-reverse (lst &optional acc)  
  (if lst  
      (alt-reverse (rest lst) (cons (first lst) acc))  
      acc  
    )  
  )
```

Reverse

Processing
before recursion
(*tail-recursion*)

```
(defun alt-reverse (lst &optional acc)
  (if lst
      (alt-reverse (rest lst) (cons (first lst) acc))
      acc)
  )
```



acc when calling (**alt-reverse** *intermediary-list*)

acc when calling (**alt-reverse** *lst*)

Nested Lists as an Inductive Data Type

- **NestedList** =
 - | **nil** | **atom** (other than **nil**)
 - | **cons** **NestedList** **NestedList** } *Type definition*
-
- nil**
(**cons** *left right*) } *Construction forms*
- (**null** **nil**) \rightarrow **t**
(**null** (**cons** *left right*)) \rightarrow **nil**
(**consp** **nil**) \rightarrow **nil**
(**consp** *atom*) \rightarrow **nil**
(**consp** (**cons** *left right*)) \rightarrow **t**
(**first** (**cons** *left right*)) \rightarrow *left*
(**rest** (**cons** *left right*)) \rightarrow *right* } *Elimination equations*

NestedList =

| nil | atom

| cons NestedList NestedList

Recursive Pattern for Nested Lists

```
(defun function-name (nested-list)  
  (cond  
    ((null nested-list) ...nil-base-case-processing...)  
    ((consp nested-list) ...induction-case-processing...  
     if needed, call (function-name (first nested-list))  
                     (function-name (rest nested-list)))  
    )  
  (otherwise ...atom-base-case-processing...)  
  )  
)
```

NestedList =

| nil | atom

| cons NestedList NestedList

Recursive Pattern for Nested Lists (2)

```
(defun function-name (nested-list)
```

```
  (if (consp nested-list)
```

```
    ...induction-case-processing...
```

*one
expression*

```
    if needed, call (function-name (first nested-list))
```

```
                    (function-name (rest nested-list))
```

*one
expression
(optional)*

```
    ...base-cases-processing...(nil, atom)
```

```
  )
```

```
)
```

NestedList =

| **nil** | **atom**

| **cons** NestedList NestedList

Count Elements of Nested Lists

```
(defun alt-count-nodes (nested-list)
  (cond ((null nested-list) 0)
        ((consp nested-list)
         (+ (alt-count-nodes (first nested-list))
            (alt-count-nodes (rest nested-list))
         )
        )
        )
  )
  )
  )
```

NestedList =

| nil | atom

| cons NestedList NestedList

Copy Nested List

```
(defun alt-copy-nested-list (nested-list)
  (if (consp nested-list)
      (cons (alt-copy-nested-list (first nested-list))
            (alt-copy-nested-list (rest nested-list)))
      nested-list)
)
```

Trees as an Inductive Data Type

- **Tree** =
 - | **nil**
 - | **list** **Tree** Element **Tree** } *Type definition*
-
- nil**
(**list** *left elm right*) } *Construction forms*
- (**null** **nil**) \rightarrow t
(**null** (**list** *left elm right*)) \rightarrow **nil**
(**first** (**list** *left elm right*)) \rightarrow *left*
(**second** (**list** *left elm right*)) \rightarrow *elm*
(**third** (**list** *left elm right*)) \rightarrow *right* } *Elimination equations*

Tree =

| nil

| list Tree Element Tree

Recursive Pattern for Trees

```
(defun function-name (tree)
```

```
  (cond
```

```
    ((null tree)    ...base-case-processing...)
```

```
    (otherwise ...induction-case-processing...
```

```
      if needed, call (function-name (first tree))
```

```
                    (function-name (third tree))
```

```
    )
```

```
  )
```

```
)
```

Tree =

| nil

| list Tree Element Tree

Recursive Pattern for Trees (2)

```
(defun function-name (tree)
```

```
  (if tree
```

```
    ...induction-case-processing...
```

*one
expression*

```
      if needed, call (function-name (first tree))
```

```
                    (function-name (third tree))
```

```
    ...base-case-processing...
```

*one
expression
(optional)*

```
  )
```

```
)
```

Tree =

| nil

| list Tree Element Tree

Count Elements of Tree

```
(defun alt-count-nodes (tree)
```

```
  (if tree
```

```
    (+ (alt-count-nodes (first tree))
```

```
       1
```

```
      (alt-count-nodes (third tree))
```

```
    )
```

```
    0
```

```
  )
```

```
)
```

Tree =

| nil

| list Tree Element Tree

Copy Tree

```
(defun alt-copy-tree (tree)
  (if tree
      (list (alt-copy-tree (first tree))
            (second tree)
            (alt-copy-tree (third tree)))
      )
  )
)
```