Advanced Functional and Logic Programming

Programming principles

Mircea Marin
mircea.marin@e-uvt.ro

November 8, 2018

Remember that ...

Recursion is a programming technique to solve problems by breaking them in one of more problems that are similar but simpler than the initial problem.

- We can define and work with
  - recursive functions, like the Fibonacci function
    ```
    (define (fib n)
       (if (= n 0) 1 (* n (fib (- n 1)))))
    ```
  - recursive datatypes, like lists of symbols:
    ```
    ⟨nlist⟩ ::= null
             | (cons ⟨symbol⟩ ⟨nlist⟩)
    ```
- In functional programming, all repetitive computations are performed by recursion
  - We can not perform usual iterative computations, because we can not change the values of variables.
  - Recursively defined functions are efficient when they are tail recursive

Structural recursion

Can be used to define functions which take one or more arguments that belong to a composite type $\langle type \rangle$:

$\langle type \rangle$ `::=` $case_1$ | ... | $case_n$

where each case $case_i$ is ($constr_i$ $\langle type_{i,1} \rangle$ ... $\langle type_{i,k_i} \rangle$)

- A recognizer of values $\text{v} \in \langle type \rangle$:

  (define ($type$? v) (or $test_1$
  ...
  $test_n$))

  where $test_i$ is of the form

  (and ($constr_i$? v) ($type_{i,1}$? ($sel_{i,1}$ v)) ... ($type_{i,k_i}$? (sel_{i,k_i} v)))

- A function with an argument $\text{v} \in \langle type \rangle$ has the form

  (define (f ... v ...)
  (cond [($constr_1$? v) ⟨computation involving ($sel_{1,1}$ v) ... (sel_{1,k_1} v)⟩]
  ...
  [($constr_n$? v) ⟨computation involving ($sel_{n,1}$ v) ... (sel_{n,k_n} v)⟩]))

Example 1: structural recursion on lists of numbers

$\langle \text{lon} \rangle$ `::= null | (cons` $\langle \text{number} \rangle$ $\langle \text{lon} \rangle$`)`

Example 1: structural recursion on lists of numbers

⟨lon⟩ `::= null | (cons` ⟨number⟩ ⟨lon⟩`)`

1. Recognizer (`lon? v`):

```
(define (lon? v)
  (or (null? v)
      (and (number? (car v)) (lon? (cdr v)))))
```

Example 1: structural recursion on lists of numbers

$\langle$lon$\rangle$ `::= null | (cons` $\langle$number$\rangle$ $\langle$lon$\rangle$`)`

1. Recognizer `(lon? v)`:

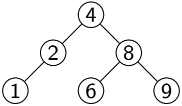
```
(define (lon? v)
  (or (null? v)
      (and (number? (car v)) (lon? (cdr v)))))
```

2. Define `(app2 lst1 lst2)` which appends the lists of numbers `ls1`, `lst2`:

```
(define (app2 lst1 lst2)
   (cond [(null? lst1) lst2]
         [(cons? lst1) (cons (car lst1)
                             (app2 (cdr lst1) lst2))]))
```

Example 2: structural recursion on binary trees of numbers

⟨nTree⟩ `::= null | (list` ⟨number⟩ ⟨nTree⟩ ⟨nTree⟩`)`



has the encoding

```
(list 4 (list 2 (list 1 null null)
                null)
        (list 8 (list 6 null null)
                (list 9 null null)))
```

1. Recognizer (nTree? v):

```
(define (nTree? v)
  (or (null? v)
      (and (list? v) (= (length v) 3)
           (number? (car v)) (nTree?  (cadr v)) (nTree?  (caddr v)))))
```

2. Define (sumNodes t) which computes the sum of numbers in all nodes of t ∈ ⟨nTree⟩:

```
(define (sumNodes t)
  (cond [(null? t) 0]
        [(list? t) (+ (car t) (sumNodes (cadr t))  (sumNodes (caddr t)))]))
```

Structural recursion on numbers

Choose the best structural description of $\mathbb{N}$

- There are many ways to define the type $\langle\text{nat}\rangle$ of natural numbers by structural recursion. For example:

  $\langle\text{nat}\rangle$ `::= 0 | (+ 1` $\langle\text{nat}\rangle$`)`

  or

  $\langle\text{nat}\rangle$ `::= 0 |` $\langle\text{even}\rangle$ `|` $\langle\text{odd}\rangle$
  $\langle\text{even}\rangle$ `::= (* 2` $\langle\text{nat}\rangle$`)`
  $\langle\text{odd}\rangle$ `::= (+ 1` $\langle\text{even}\rangle$`)`

  or . . .

- Different choices affect the efficiency of the recursive function (see next slide).

Recursive functions with numeric arguments

Example: computation of x^n for $n \in \mathbb{N}$

1. Version 1: $x^n = \begin{cases} 1 & \text{if } n = 0, \\ x \cdot x^{n-1} & \text{if } n > 0. \end{cases}$

```
(define (expt-v1 x n)
   (if (= n 0) 1 (* x (expt-v1 x (- n 1)))))
```

2. Version 2: $x^n = \begin{cases} 1 & \text{if } n = 0, \\ (x^2)^{n/2} & \text{if } n \text{ is even}, \\ x * (x^2)^{(n-1)/2} & \text{if } n \text{ is odd}. \end{cases}$

```
(define (expt-v2 x n)
   (cond [(= n 0) 1]
         [(even? n) (expt-v2 (* x x) (/ n 2))]
         [(odd? n) (* x (expt-v2 (* x x) (/ (- n 1) 2)))]))
```

Solving a more general problem

Sometimes, the best way to solve a problem is to solve a more general problem and use it to solve the original problem as a special case.

Example

Suppose `von` is a vector of numbers. Define `(vector-sum von)` which returns the sum of elements of `von`, using the functions

- `(vector-length von)`: returns the length (number of elements) of `von`
- `(vector-ref von i)`: returns the `i`-th element of `von`; the elements of `von` are indexed starting from 0.

Solving a more general problem

Sometimes, the best way to solve a problem is to solve a more general problem and use it to solve the original problem as a special case.

Example

Suppose `von` is a vector of numbers. Define `(vector-sum von)` which returns the sum of elements of `von`, using the functions

- `(vector-length von)`: returns the length (number of elements) of `von`
- `(vector-ref von i)`: returns the `i`-th element of `von`; the elements of `von` are indexed starting from 0.

Instead of defining `(vector-sum von)`, we can define the **more general function** `(partial-vector-sum von n)` which computes the sum of elements with indexes from 0 to `n` in `von`.

Solving a more general problem

Sometimes, the best way to solve a problem is to solve a more general problem and use it to solve the original problem as a special case.

Example

Suppose `von` is a vector of numbers. Define (`vector-sum von`) which returns the sum of elements of `von`, using the functions

- (`vector-length von`): returns the length (number of elements) of `von`
- (`vector-ref von i`): returns the `i`-th element of `von`; the elements of `von` are indexed starting from 0.

Instead of defining (`vector-sum von`), we can define the **more general function** (`partial-vector-sum von n`) which computes the sum of elements with indexes from 0 to `n` in `von`.

- Note that (`vector-sum von`) coincides with (`partial-vector-sum von (vector-length von)`)

Solving a more general problem

Example: compute the sum of elements of a vector of numbers

```
(define (vector-sum von)
   (define (partial-vector-sum von n)
      (if (= n 0)
         0  ; nothing to add
         (+ (partial-vector-sum von (- n 1))
             (vector-ref von (- n 1)))))
   (partial-vector-sum von (vector-length von)))
```

Remarks

1. `partial-vector-sum` is an auxiliary function used only in the implementation of `vector-sum`
   - ⇒ we can make the definition of `partial-vector-sum` local to the body of `vector-sum`
2. `von` refers to the same vector of numbers in all function calls
   - ⇒ The formal argument `von` can be removed from the definition of `partial-vector-sum`

Solving a more general problem

Example: compute the sum of elements of a vector of numbers

```
(define (vector-sum von)
  (define (partial-vector-sum von n)
    (if (= n 0)
      0  ; nothing to add
      (+ (partial-vector-sum von (- n 1))
         (vector-ref von (- n 1)))))
  (partial-vector-sum von (vector-length von)))
```

NOTE: This simplified implementation is more efficient.

Linear recursive functions

A function $f : \mathbb{N} \to \mathbb{N}$ is **linear recursive** if

- We know the first k values of f:

$$f(0) = f_0, f(1) = f_1, \ldots, f(k-1) = f_{k-1}$$

- We know $a_1, \ldots, a_k$ such that, for all $n \geq k$:

$$f(n) = a_1 \cdot f(n-1) + a_2 \cdot f(n-2) + \ldots + a_k \cdot f(n-k)$$

Linear recursive functions

A function $f : \mathbb{N} \to \mathbb{N}$ is **linear recursive** if

- We know the first k values of f:

$$f(0) = f_0, f(1) = f_1, \ldots, f(k-1) = f_{k-1}$$

- We know $a_1, \ldots, a_k$ such that, for all $n \geq k$:

$$f(n) = a_1 \cdot f(n-1) + a_2 \cdot f(n-2) + \ldots + a_k \cdot f(n-k)$$

All linear recursive functions have a tail recursive definition.
(see next slide)

Linear recursive functions

A tail recursive definiton

Assume $f : \mathbb{N} \to \mathbb{N}$ is **linear recursive**:

- $f(0) = f_0, f(1) = f_1, \ldots, f(k-1) = f_{k-1}$
- $f(n) = a_1 \cdot f(n-1) + a_2 \cdot f(n-2) + \ldots + a_k \cdot f(n-k)$ for all $n \geq k$

Linear recursive functions
A tail recursive definiton

Assume $f : \mathbb{N} \to \mathbb{N}$ is **linear recursive**:

- $f(0) = f_0, f(1) = f_1, \ldots, f(k-1) = f_{k-1}$
- $f(n) = a_1 \cdot f(n-1) + a_2 \cdot f(n-2) + \ldots + a_k \cdot f(n-k)$ for all $n \geq k$

Then

$$f(n) = \textit{f-acc}(n, f_0, f_1, \ldots, f_{k-1})$$

where $\textit{f-acc}(n, A_0, A_1, \ldots, A_{k-1})$ is

- A_n if $0 \leq n < k$, (the base cases)
- $\textit{f-acc}(A_1, \ldots, A_{k-1}, a_1 \cdot A_{k-1} + a_2 \cdot A_{k-2} + \ldots + a_k \cdot A_0)$, otherwise.

PROOF:

$\textit{f-acc}(n, f_0, f_1, \ldots, f_{k-2}, f_{k-1}) =$
$\textit{f-acc}(n, f(0), f(1), \ldots, f(k-2), f(k-1)) =$
$\textit{f-acc}(n-1, f(1), f(2), \ldots, f(k-1), f(k)) =$
$\ldots$
$\textit{f-acc}(k-1, f(n-k+1), f(n-k+2), \ldots, f(n-1), f(n)) =$
$f(n)$

Primitive recursive functions

A definition by primitive recursion of h : $\mathbb{N} \times A_1 \times \ldots \times A_k \to B$ looks as follows:

1. $h(0, x_1, \ldots, x_k) := f(x_1, \ldots, x_k)$ (base case)
2. $h(n+1, x_1, \ldots, x_k) := g(n, h(n, x_1, \ldots, x_k), x_1, \ldots, x_k)$ for all $n \in \mathbb{N}$. (recursive case)

Remark

Every primitive recursive function h has a tail recursive definition.

```
(define (h n x1 ... xk)
  (define (h-acc i prev)
    (if (= i n)
        prev
        (h-acc (+ i 1) (g i prev x1 ... xk))))
  (h-acc 0 (f x1 ... xk)))
```

apply and map

$\langle\text{list}\rangle$ `::= null | (cons` $\langle\text{value}\rangle$ $\langle\text{list}\rangle$`)`

- If `f` is a function and `lst` a list of values $v_1, \ldots, v_n$ then

  `(apply f lst)`

  returns the value of the function call $(f\ v_1\ \ldots\ v_n)$
- If `f` is a function which expects `n` arguments, and

$$
\begin{array}{rl}
l_1 & = (\texttt{list}\ v_{11}\ v_{12}\ \ldots\ v_{1k}) \\
\ldots & \\
l_n & = (\texttt{list}\ v_{n1}\ v_{12}\ \ldots\ v_{nk}) \\
\text{then } (\texttt{map}\ f\ l_1\ \ldots\ l_n) & = (\texttt{list}\ v_1\ \ v_2\ \ \ldots\ \ v_k) \\
\text{where } v_1 & = (f\ v_{11}\ v_{21}\ \ldots\ v_{n1}) \\
v_2 & = (f\ v_{12}\ v_{22}\ \ldots\ v_{n2}) \\
 & \ldots \\
v_k & = (f\ v_{1k}\ v_{2k}\ \ldots\ v_{nk})
\end{array}
$$

apply and map
Examples

```
> (apply + '(1 2 3 4)) ; compute 1+2+3+4
10
> (apply append '((1 2) (a b) () (3 4)))
'(1 2 a b 3 4)
> (map (lambda (x) (* x x)) '(1 2 3))
'(1 4 9)
> (map cons '(a b c) '(1 2 3))
'((a . 1) (b . 2) (c . 3))
> (map reverse '((a b c) (#t #f) ((1 2) (3 4))))
'((c b a) (#f #t) ((3 4) (1 2)))
```

REMARK: `(reverse lst)` returns the list of elements of `lst` in reverse order: It can be defined by tail recursion:

```
(define (reverse lst)
   (define (reverse-acc lst acc) ; reverse-acc is tail recursive
      (if (null? lst)
          acc
          (reverse-acc (cdr lst) (cons (car lst) acc))))
   (reverse-acc lst '()))
```

filter, foldl, foldr

1. If p is a boolean function, and lst is a list, then

   `(filter p lst)`

   returns lst without the elements v for which (p v) is #f.

2. If $f : A \times B \to B$ is a binary function, $b \in B$, and lst is a list of values $a_1, \ldots, a_n \in B$, then

   `(foldl f b lst)`

   returns the value of $(f(a_n, \ldots (f(a_1, b)) \ldots)$.
   If lst is '(), the returned value is b.

3. If $f : A \times B \to B$ is a binary function, $b \in B$, and lst is a list of values $a_1, \ldots, a_n \in B$, then

   `(foldr f b lst)`

   returns the value of $(f(a_1, \ldots (f(a_n, b)) \ldots)$.
   If lst is '(), the returned value is b.

filter,foldl,foldr

Examples

```
> (filter symbol? '(a 1 (1 . 2) bc "bc" #t ()))
'(a bc)
> (filter (lambda (x) (and (list? x)
                           (= (length x) 2)
                           (number? (car x))
                           (number? (cadr x))))
      '((1 . 2) (4 3) #(1 2) #t abc (3 2) (1 2 3)))
'((4 3) (3 2))
> (foldl (lambda (b a) (list b a)) 'a '(b1 b2 b3))
'(b3 (b2 (b1 a)))
> (foldr (lambda (b a) (list b a)) 'a '(b1 b2 b3))
'(b1 (b2 (b3 a)))
> (foldl cons '() '(a b c))
'(c b a)
> (foldr cons '() '(a b c))
'(a b c)
```

Properties of filter,foldl,foldr

filter, foldl and foldr are predefined functions.
We can define them recursively:

```
(define (filter p lst)
  (cond [(null? lst) null]
        [(p (car lst)) (cons (car lst) (filter p (cdr lst)))]
        [#t (filter p (cdr lst))]))
(define (foldl f b lst)
   (cond [(null? lst) b]
         [#t (foldl f (f (car lst) b) (cdr lst))]))
(define (foldr f b lst)
   (cond [(null? lst) b]
         [#t (f (car lst) (foldr f b (cdr lst)))]))
```

1. foldl is tail recursive → efficient implementation
2. foldr is not tail recursive, but note that:
   - (foldr f b lst)=(foldl f b (reverse lst))

   ⇒ a better definition of foldr, which is tail-recursive:

   ```
   (define (foldr f b lst) (foldl f b (reverse lst))
   ```

Remarks

Several real-world applications operate on large collections of data that can be implemented as lists, and the operations on such collections are compositions of a small number of generic operations.

- This observation led to the idea of applicative programming with lists, where all operations of practical interest are defined as combinations of a small number of generic operations, such as

  ```
  map
  apply
  filter
  foldl
  foldr
  reverse
  ```

- Applicative programming with lists inspired the MapReduce programming model used by Google to process and generate large data sets.

Illustrated example

A database of employees

Database: list of records of the form `(list` *name salary job*`)`

```
> (define employees
'(("John White" 10000 "manager") ("Sam Smith" 4500 "cook")
  ("Alice Cooper" 3600 "secretary") ("Ray Ban" 6320 "driver")
  ("Mike Cole" 2600 "waiter") ("Ana Fox" 3800 "secretary")
  ("John Black" 2450 "driver") ("Jack White" 3500 "cook")))
```

Illustrated example

A database of employees

Database: list of records of the form (`list` *name salary job*)

```
> (define employees
'(("John White" 10000 "manager") ("Sam Smith" 4500 "cook")
  ("Alice Cooper" 3600 "secretary") ("Ray Ban" 6320 "driver")
  ("Mike Cole" 2600 "waiter") ("Ana Fox" 3800 "secretary")
  ("John Black" 2450 "driver") ("Jack White" 3500 "cook")))
```

O_1) Get the records of people with salary greater that 4000:

```
> (filter (lambda (emp) (> (cadr emp) 4000)) employees)
'(("John White" 10000 "manager")
  ("Sam Smith" 4500 "cook")
  ("Ray Ban" 6320 "driver"))
```

Illustrated example
A database of employees

Database: list of records of the form (`list` *name salary job*)

```
> (define employees
'(("John White" 10000 "manager") ("Sam Smith" 4500 "cook")
  ("Alice Cooper" 3600 "secretary") ("Ray Ban" 6320 "driver")
  ("Mike Cole" 2600 "waiter") ("Ana Fox" 3800 "secretary")
  ("John Black" 2450 "driver") ("Jack White" 3500 "cook")))
```

O_1) Get the records of people with salary greater that 4000:

```
> (filter (lambda (emp) (> (cadr emp) 4000)) employees)
'(("John White" 10000 "manager")
  ("Sam Smith" 4500 "cook")
  ("Ray Ban" 6320 "driver"))
```

O_2) Get a digest with the names of all people employed in the company

```
> (map car employees)
'("John White" "Sam Smith" "Alice Cooper" "Ray Ban"
  "Mike Cole" "Ana Fox" "John Black" "Jack White")
```

Illustrated example

A database of employees

Database: list of records of the form (`list` *name salary job*)

```
> (define employees
'(("John White" 10000 "manager") ("Sam Smith" 4500 "cook")
  ("Alice Cooper" 3600 "secretary") ("Ray Ban" 6320 "driver")
  ("Mike Cole" 2600 "waiter") ("Ana Fox" 3800 "secretary")
  ("John Black" 2450 "driver") ("Jack White" 3500 "cook")))
```

O_1) Get the records of people with salary greater that 4000:

```
> (filter (lambda (emp) (> (cadr emp) 4000)) employees)
'(("John White" 10000 "manager")
  ("Sam Smith" 4500 "cook")
  ("Ray Ban" 6320 "driver"))
```

O_2) Get a digest with the names of all people employed in the company

```
> (map car employees)
'("John White" "Sam Smith" "Alice Cooper" "Ray Ban"
  "Mike Cole" "Ana Fox" "John Black" "Jack White")
```

O_3) Total amount of money spent to pay the employees' salaries:

```
> (foldl (lambda (emp b) (+ (cadr emp) b)) 0 employees)
36770
```

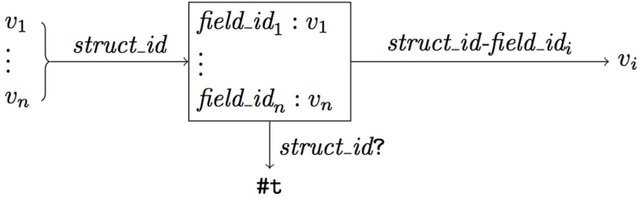
The struct special form

Users can define their own composite types with the special form struct:

(struct *struct_id* (*field_id*$_1$ ... *field_id*$_n$))

creates the following functions for the newly declared type *struct_id*:

- the **constructor** *struct_id* that takes as many arguments as the number of fields
- the **recognizer** *struct_id*?
- n **selectors** *struct_id*–*field_id*$_i$, for each of the fields of the new composite type



Structures
Examples

```
> (struct emp (name salary job))
> ; create an instance of emp
  (define e1 (emp "Ana Schwarz" 2300 "attorney"))
> (emp? e1)
#t
> (list (emp-name e1) (emp-salary e1))
'("Ana Schwarz" 2300)
```

- By default, the values of structures are opaque:
  ```
  > e1
  #<emp>
  ```
- We can define structures with transparent values if we use the `#:transparent` keyword.

Example

```
> (struct emp (name salary job) #:transparent)
> (define e2 (emp "Bruce Willis" 25000 "actor"))
> e2
(emp "Bruce Willis" 25000 "actor")
```

A convenient abbreviation

Many datatypes (including pairs, lists, vectors) have quoted values of the form *'datum*

Usually, these forms are valid input forms.

Examples

```
> '(1 2 (3 4))   ; shorter input than (list 1 2 (list 3 4))
'(1 2 (3 4))
> '#(1 2 (3 4)) ; shorter input than (vector 1 2 (list 3 4))
'#(1 2 (3 4))
>'(a . #(b c))   ; shorter input than (cons 'a (vector 'b 'c))
'(a . #(b c))
```

Bad news: quoted expressions **can not** be used to create composite values from component values.

Good news: quasiquoted expressions **can** be used to create composite values from component values (see next slide).

Quasiquoted expresions

- A **quasiquoted expression** is of the form

  ` *datum*

  It is like a quoted expression, but is starts with the character `

- Inside a quasiquoted expression, every subexpression of the form

  , *expr*

  is replaced by the value of *expr*

- Inside a quasiquoted expression, every subexpression of the form

  ,@*expr*

  where the value of *expr* is a list of values $v_1, \ldots, v_n$, is replaced by the sequence of values $v_1 \ \ldots \ v_n$

Quasiquoted expressions
Examples

```
> (define a '(Toyota Prius))
> (define b 2011)
> (define c "red")
> ; a quasiquoted list
  `(car (model ,@a) (year ,b) (color ,c))
'(car (model Toyota Prius) (year 2011) (color "red"))
> `#(,@a is ,c) ; a quasiquoted vector
'#(Toyota Prius is "red")
```

Quasiquoted expressions
Examples

```
> (define a '(Toyota Prius))
> (define b 2011)
> (define c "red")
> ; a quasiquoted list
  `(car (model ,@a) (year ,b) (color ,c))
'(car (model Toyota Prius) (year 2011) (color "red"))
> `#(,@a is ,c) ; a quasiquoted vector
'#(Toyota Prius is "red")
```

Question

What is the following function doing when `lst` is a list of integers?

```
(define (del2 lst)
  (if (null? lst) '()
      (if (even? (car lst))
          `(,@(del2 (cdr lst)))
          `(,(car lst) ,@(del2 (cdr lst))))))
```

A general pattern of recursion

This is a frequent pattern to transform composite values:

```
(define (transform cv)
  (cond [base-case₁ value₁]
        ...
        [base-caseₘ valueₘ]
        [recursive-case₁
          ; compute (transform v₁) ... (transform vₙ)
          ; for the component values v₁, ..., vₙ of cv
          ; and combine them into a return value
        ]
        [recursive-case₂ ...]
        ...))
```

Example

Compute the flattened form of a nestes list of symbols sl $\in \langle \mathrm{S}-\mathrm{list}\rangle$

$$\langle \mathrm{S}-\mathrm{list}\rangle ::= \langle \mathrm{symbol}\rangle \mid (\texttt{list}\ \langle \mathrm{S}-\mathrm{list}\rangle\ \ldots\ \langle \mathrm{S}-\mathrm{list}\rangle)$$

The flattened form of an S-list sl is defined as follows:

- (list sl) if sl is a symbol.
- Otherwise, sl $=$ (list $\mathtt{sl}_1\ \ldots\ \mathtt{sl}_n$) and the flattened form is the result of appending the flattened forms of $\mathtt{sl}_1, \ldots, \mathtt{sl}_n$ in this order.

Example

```
> (flatten-list '(((a) (b (() c d)) e f ((g)))))
'(a b c d e f g)
```

Example

Compute the flattened form of a nestes list of symbols `sl` $\in \langle \mathrm{S-list} \rangle$

$\langle \mathrm{S-list} \rangle$::=$\langle \mathrm{symbol} \rangle$ | (list $\langle \mathrm{S-list} \rangle$ ... $\langle \mathrm{S-list} \rangle$)

The flattened form of an S-list `sl` is defined as follows:

- (list sl) if `sl` is a symbol.
- Otherwise, $\mathtt{sl} = (\mathtt{list}\ \mathtt{sl}_1\ \ldots\ \mathtt{sl}_n)$ and the flattened form is the result of appending the flattened forms of $\mathtt{sl}_1, \ldots, \mathtt{sl}_n$ in this order.

Example

```
> (flatten-list '(((a) (b (() c d)) e f ((g)))))
'(a b c d e f g)
```

Implementation

```
(define (flatten-list sl)
  (cond [(symbol? sl) (list sl)]
        [#t (apply append (map flatten-list sl))]))
```

An illustrated example

Inward propagation of negation in propositional formulas

```
⟨prop⟩ ::= ⟨symbol⟩                      ; atomic formula
         | (list 'not ⟨prop⟩)            ; negation
         | (list 'and ⟨prop⟩ ⟨prop⟩)     ; conjunction
         | (list 'or  ⟨prop⟩ ⟨prop⟩)     ; disjunction
```

- If cv $= \neg(P \vee Q)$ then $\mathit{transf}(\mathtt{cv}) = \mathit{transf}(\neg P) \wedge \mathit{transf}(\neg Q)$.
- If cv $= \neg(P \wedge Q)$ then $\mathit{transf}(\mathtt{cv}) = \mathit{transf}(\neg P) \vee \mathit{transf}(\neg Q)$.
- If cv $= \neg(\neg P)$ then $\mathit{transf}(\mathtt{cv}) = \mathit{transf}(P)$.
- If cv is an atom, then $\mathit{transf}(\mathtt{cv}) = \mathtt{cv}$.
- If cv is $P \vee Q$ then $\mathit{transf}(\mathtt{cv}) = \mathit{transf}(P) \vee \mathit{transf}(Q)$.
- If cv is $P \wedge Q$ then $\mathit{transf}(\mathtt{cv}) = \mathit{transf}(P) \wedge \mathit{transf}(Q)$.
- If cv is $\neg P$ where P is not negation, then $\mathit{transf}(\mathtt{cv}) = \neg \mathit{transf}(Q)$.

Top down transformation of composite values

Example: inward propagation of negation in propositional formulas (contd.)

- Useful recognizers for all kinds of propositional formulas:

```
(define (atom? f)
    (symbol? f))
(define (not? f)
   (and (list? f) (= (length f) 2) (eq? (car f) 'not)))
(define (and? f)
   (and (list? f) (= (length f) 3) (eq? (car f) 'and)))
(define (or? f)
   (and (list? f) (= (length f) 3) (eq? (car f) 'or)))
```

Top down transformation of composite values

Example: inward propagation of negation in propositional formulas (contd.)

- Definition of (propagate-not cv)

```
(define (propagate-not cv)
  (cond [(atom? cv) cv]
        [(and (not? cv) (or? (cadr cv)))
         (let ([P (list-ref (cadr cv) 1)]
               [Q (list-ref (cadr cv) 2)])
           `(and ,(propagate-not `(not ,P))
                 ,(propagate-not `(not ,Q))))]
        [(and (not? cv) (and? (cadr cv)))
         (let ([P (caddr cv)]
               [Q (cadddr cv)])
           `(or ,(propagate-not `(not ,P))
                ,(propagate-not `(not ,Q))))]
        [(and (not? cv) (not? (cadr cv)))
         (propagate-not (list-ref (cadr cv) 1))]
        [#t `(,(car cv) ,@(map propagate-not (cdr cv)))]))
```