Examples of how a Functional Program can be Developed

From

- an existing recursive program
- analysis of input and output diagrams

© Gunnar Gotshalks

Transpose a 2d-Matrix – 1

- 2-d matrix is represented as a list of rows all of the same length
- For example

```
1 2 3
4 5 6      -->   ((1 2 3)(4 5 6)(7 8 9))
7 8 9
```

- The transpose (swap rows and columns) of the above is

```
1 4 7
2 5 8      -->   ((1 4 7)(2 5 8)(3 6 9))
3 6 9
```

© Gunnar Gotshalks

Transpose a 2d-Matrix – 2

```
(defun trans ( theMatrix )
  (cond ( ( null ( car  theMatrix ) ) nil )
        ( t ( cons ( firstOfEach  theMatrix )
                   ( trans  (restOfEach  theMatrix ))))
))

(defun firstOfEach ( theMatrix )  ; Extract first of each row
  (cond ( ( null theMatrix ) nil )
        ( t  (cons ( caar  theMatrix )
                   ( firstOfEach  ( cdr  theMatrix ))))
))

(defun restOfEach ( theMatrix ) ; remove first of each row
  (cond ( ( null  theMatrix ) nil )
        ( t  ( cons  ( cdar  theMatrix )
                     ( restOfEach  ( cdr  theMatrix ))))
))
```

© Gunnar Gotshalks

Transpose a 2d-Matrix – 3

- Analysis of the transpose program shows that **trans** invokes **firstOfEach** to every decreasing rows (**restOfEach**)
- This is what maplist does
- So a first pass of trans becomes

```
(defun trans (theMatrix)
  (maplist 'firstOfEach theMatrix)
)
```

  - » **(trans '((1 2 3) (4 5 6) (7 8 9))) ==> ((1 4 7) (4 7) (7))**

- What went wrong?

© Gunnar Gotshalks

Transpose a 2d-Matrix – 4

- Put a print statement in **firstOfEach**

```
(defun firstOfEach (theMatrix)  ; Extract first of each row
 (print theMatrix)
  (cond ((null theMatrix) nil)
     (t (cons (caar theMatrix) (firstOfEach (cdr theMatrix))))
))
```

- The output is

```
((1 2 3) (4 5 6) (7 8 9))    ; first call from maplist
((4 5 6) (7 8 9))            ; recursion
((7 8 9))
NIL
((4 5 6) (7 8 9))            ; second call from maplist
((7 8 9))                    ; recursion
NIL
((7 8 9))                    ; third call from maplist
NIL                          ; recursion
((1 4 7) (4 7) (7))          ; the answer
```

© Gunnar Gotshalks

Transpose a 2d-Matrix – 5

- **maplist** is removing the rows not the first of each row because **maplist** is working on the matrix a row at a time
  - **Input is ((1 2 3) (4 5 6) (7 8 9)) -- one list of rows**
- We want maplist to work on each row
  - **Input should be (1 2 3) (4 5 6) (7 8 9) -- three lists**
  - **This is a common problem we want to remove the outer parenthesis**
  - **Recall that apply removes the outer level of parenthesis when invoking a function on arguments**
- Thus **trans** becomes

```
(defun trans (theMatrix)
  (apply ‘maplist ‘firstOfEach theMatrix)
)
```

© Gunnar Gotshalks

Transpose a 2d-Matrix – 6

- We try **trans** and get an error message such as

  Error: Expected 1 args but received 3 args

  Fast links are on: do (si::use-fast-links nil) for debugging

  Error signalled by MAPLIST.

  Broken at FIRSTOFEACH

© Gunnar Gotshalks

Transpose a 2d-Matrix – 7

- Ah! now we have one argument for each row as input to **firstOfEach** but the function expects a single argument – a list of rows
  - » **Use the keyword &rest to gather all the arguments into one.**

```
(defun firstOfEach  ( &rest  theMatrix )
 ( cond ( ( null theMatrix ) nil )
        ( t  ( cons  ( caar  theMatrix )
                     ( firstOfEach  ( cdr  theMatrix ))))
))
```

© Gunnar Gotshalks

Transpose a 2d-Matrix – 8

- We try **trans** and get infinite recursion – the print statement shows the following for the first few lines

```
((1 2 3) (4 5 6) (7 8 9))
(((4 5 6) (7 8 9)))          ; list nested one deeper
(NIL)
(NIL)
(NIL)   goes on forever
```

© Gunnar Gotshalks

Transpose a 2d-Matrix – 9

- Each recursive call to **firstOfEach** adds a layer of parenthesis
  - **Again a common error – we need to remove the parenthesis before the recursive call – use apply**

```
(defun firstOfEach  ( &rest  theMatrix )
 ( cond ( ( null theMatrix ) nil )
        ( t  ( cons  ( caar  theMatrix )
                     ( apply 'firstOfEach
                             ( cdr  theMatrix ))))
))
```

© Gunnar Gotshalks

Transpose a 2d-Matrix – 10

- **trans** now works with the upper level being a functional but **firstOfEach** is still recursive

```
(defun  trans  ( theMatrix )
  ( apply  'maplist  'firstOfEach  theMatrix )
)

(defun  firstOfEach  ( &rest   theMatrix )
 ( cond  ( ( null  theMatrix ) nil )
         ( t  ( cons  ( caar  theMatrix )
                      ( apply  'firstOfEach
                               ( cdr  theMatrix ))))
))
```

© Gunnar Gotshalks

Transpose a 2d-Matrix – 11

- Notice that **firstOfEach** takes the first item from each sublist

```
(defun firstOfEach (&rest  theMatrix)
 (cond ((null theMatrix) nil)
       (t (cons (caar theMatrix) (apply 'firstOfEach
                                        (cdr theMatrix))))
))
```

- **car** gives the first of a list and **mapcar** will apply it to every sublist in a list and collect the results in a list so we have

```
(defun  firstOfEach  ( &rest  theMatrix )
 ( mapcar  'car  theMatrix )
)
```

© Gunnar Gotshalks

Transpose a 2d-Matrix – 12

- We have two functionals for the solution

```
(defun  trans  ( theMatrix )
  ( apply  'maplist  'firstOfEach  theMatrix )
)

(defun  firstOfEach  ( &rest  theMatrix )
  ( mapcar  'car  theMatrix )
)
```

- Using lambda we can eliminate **firstOfEach**

```
(defun  trans  ( theMatrix )
  (apply  'maplist   #'( lambda  ( &rest  theMatrix )
                                 ( mapcar  'car  theMatrix ))
          theMatrix )
)
```

© Gunnar Gotshalks

Transpose a 2d-Matrix – 13

- But nothing beats creative insight and knowledge of available operations
- The following gives the transpose

```
(defun  trans  ( theMatrix )
  (apply  'mapcar  'list  theMatrix )
)
```

© Gunnar Gotshalks

All pairs functional – 1

- We want the following functional

  allPairs : < <a, b, c> <1, 2, 3, 4> > **input**

  ==>

  < <a,1> <a,2> <a,3> <a,4> **output**

  <b,1> <b,2> <b,3> <b,4>

  <c,1> <c,2> <c,3> <c,4> >

- We make use of the ‘picture’ of the input and output to infer a functional solution

© Gunnar Gotshalks

All pairs functional – 2

- Looking at the functionals in the library it seems that distribution may be useful
- Lets try it

  distl : < <a, b, c> <1, 2, 3, 4> >

  ==>

  < << a, b, c >, 1 > << a, b, c >, 2 > << a, b, c >, 3 > ... >

- Looks good but we want to distribute second argument over the first
- **rev** could be used but we have **distr**

  distr : < <a, b, c> <1, 2, 3, 4> >

  ==>

  < < 1, < a, b, c > > < 2, < a, b, c > > < 3, < a, b, c > ... >

© Gunnar Gotshalks

All pairs functional – 3

- We have

  distr : < <a, b, c> <1, 2, 3, 4> >

  ==>

  < < 1, < a, b, c > > < 2, < a, b, c > > < 3, < a, b, c > ... >

- If we distribute ‘right’ the numbers over each list we have

  < < <a , 1>, < b, 1>, <c, 1 > > ... >

- But examining the output we see that ‘a’ is repeated first not the “1”

  < <a,1> <a,2> <a,3> <a,4>

  <b,1> <b,2> <b,3> <b,4>

  <c,1> <c,2> <c,3> <c,4> >

output

© Gunnar Gotshalks

All pairs functional – 4

- What we need to do is to reverse the order of the arguments so the letters are distributed first

  distr o [2 , 1] : < <a, b, c> <1, 2, 3, 4> >

  ==>

  < < a, < 1, 2, 3, 4 > > < b, < 1, 2, 3, 4 > > ... >

- Now if we apply distribute left to each sublist we have

  (□ distl) : < < a, < 1, 2, 3, 4 > >
  < b, < 1, 2, 3, 4 > > ... >

  ==>

  < < < a, 1 > < a, 2 > < a, 3 > < a, 4 > >
  < < b, 1 > ... >

© Gunnar Gotshalks

All pairs functional – 5

- So far we have

  (□ distl) o distr o [2 , 1]

  ==>

  < < < a, 1 > < a, 2 > < a, 3 > < a, 4 > > < < b, 1 > ... >

- But we have the pairs nested within an extra pair of lists
- What we need to do is to **reduce** the lists into one using **append**

  (/ append) :

  < < < a, 1 > < a, 2 > < a, 3 > < a, 4 > > < < b, 1 > ... >

  ==>

  < < a, 1 > < a, 2 > < a, 3 > < a, 4 > < b, 1 > ... >

© Gunnar Gotshalks

All pairs functional – 6

- So the final function definition is

  allPairs ::= (/ append) o (□ distl) o distr o [2 , 1]

- Other orderings are possible using other combinations of swapping or not swapping the initial lists and using left or right distribution for the second distribution

  allPairs ::= (/ append) o (□ distr) o distr o [2 , 1]

  allPairs ::= (/ append) o (□ distl) o distr

  allPairs ::= (/ append) o (□ distr) o distr

© Gunnar Gotshalks