COMP0020 Functional Programming

1. Evaluating arithmeti

2. lists as functi

Contents

Programming examples.

1. writing an evaluator for arithmetic ex
   - using algebraic types and recursion
2. lists as functions
   - using higher order functions

Programming Example 1

- Writing an evaluator for arithmetic e
- Motivation :
  - a common style of programming
  - a good example of using algebraic types and recursion

Specification

- Write a Miranda program that :
  - takes as input a representation of a si ar given on the next page, and
  - returns as output the value of that expression

Grammar (BNF)

- expression : : constant | expression op ex
- op : : '*' | '+' | '-' | '/'

Preliminaries

- Note that the specification asks for a “re t

- Therefore, we can ignore lexical anal
- We will also assume that parsing has been done, so we have the synta

- We can choose an algebraic type as our representation of that synt riate constructors

Algebraic types

$$expression ::= Constant\ num$$
$$\mid App\ expression\ operator\ expression$$

$$operator ::= Times \mid Plus \mid$$

Compare with the BNF :

- expression : : constant | expression op expression | '(' expression ')'
- op : : '*' | '+' | '-' | '/'

Test values

$$\|\ (4)\ *\ 5$$

$$test1 = App\ (Bracketed\ (Constant\ 4))\ Times\ (Constant\ 5)$$

$$\|\ (4\ +\ 5)$$

$$test2 = App$$

$$(Bracketed\ (App\ (Const \quad nstant\ 5)))$$

$$Times$$

$$(Bracketed\ (App\ (Constant\ 3)\ Minus\ (Constant\ 2)))$$

Evaluator code


```
eval :: expression -> num
eval (Constant x)        = x
eval (App e1 op e2)      = evalop op (eval e1) (eval e2)
eval (Brac

evalop :: operator -> num -> num -> num
evalop Times x y         = x *
evalop Plus x y          = x + y
evalop Minus x y         = x - y
evalop Divide x y        = x / y
```

Programming Example 2

- Lists as functions
- Motivation :
  - Better understanding of, and facility with, higher order functi
  - Example of how data can be implemented as a function (a “trick”

Preamble (1)

- Recall CURRIED functions :
  - f a b c = (a + b)
  - Can help to think of binary tree giving syntax of the function applied to its arguments :

Preamble (2)

- Recall HIGHER ORDER functions :
  - f a b c = c (a b)
  - c must be a function (of at least one argument)
  - a must be a function (of at least one argument)
  - e.g. f (*2) 4 (+1)
    = (+1) ((*2) 4
    = 1 + (2 * 4)

Preamble (3)

- Recall HIGHER ORDER functions can return functions :
  - g a b = (+ a)
    main = g 3 4 5
  - Too many args ?
  - No !

Example 2 : Lists as Functions

- We have already seen how the list (1 : (2 : (3 : []))) can be represented as
  - Cons 1 (Cons 2 (Cons 3 Nil))
  - Where Cons and Nil are construct
  - The difference being that this data s ylist num and not of type **[num]**
  - This assumes the algebraic type de
    mylist * : := Nil | Cons * (mylist *)

- Now we consider a new representation :
  - cons 1 (cons 2 (cons 3 nil))
  - where cons and nil are functions ! (as follows :)

Example 2 : Lists as Functions

- The list (1 : (2 : (3 : []))) can be represented as
  - cons 1 (cons 2 (cons 3 nil))
  - where cons and nil are functions as follows :

cons a b f

nil *f* *“tail of nil”) True*

- x = cons 'A' nil is a partial application !
- y = nil is a partial application !
- Both cons and nil are partially applied — the final argument is only supplied when an element is selected or we test for nil
- To see how it works, consider the definitions of head, tail and isnil

$$
\begin{array}{llllll}
 & & & head & tail & isnil \\
 & & & \downarrow & \downarrow & \downarrow \\
cons\ a\ b\ f & = & f & a & b & False \\
nil\ \ \ \ \ \ f & = & f & (error\ ''head\ of\ nil'') & (error\ ''tail\ of\ nil'') & True
\end{array}
$$

$$
\begin{array}{lll}
head\ x & = & x\ h \\
 & & where \\
 & & h\ a\ b\ c\ =\ a \\
tail\ x & = & x\ t \\
 & & where \\
 & & t\ a\ b\ c\ =\ b \\
isnil\ x & = & x\ g \\
 & & where \\
 & & g\ a\ b\ c\ =\ c
\end{array}
$$

$Example:$

$$
\begin{array}{l}
z = cons\ 'A'\ nil \\
\ \ = cons\ 'B'\ z
\end{array}
$$

$$
\begin{array}{ll}
|| & head\ y \\
|| & \rightarrow (cons\ 'B'\ z)\ h \\
|| & \rightarrow cons\ 'B'\ z\ h \\
|| & \rightarrow h\ 'B'\ z\ False \\
|| & \rightarrow 'B'
\end{array}
$$

- Consider :
  head (tail (tail (cons a (cons b nil))))
  → (tail (tail (cons a (cons b nil)))) h
  → ((tail (cons a (cons b nil)) t) h
  → (((cons a (cons b nil)) t) t) h
  → ((cons a (cons b nil) t) t) h
  → ((t a (cons b nil) False) t) h
  → ((cons b nil) t) h
  → (cons b nil t) h
  → (t b nil False) h
  → nil h
  → h (error "head of nil") (error "tail of nil") True
  → error "head of nil"

- Consider :

  isnil nil
  → nil g
  → g (error “head of nil") (error “tail of nil") True
  → True

  isnil (cons a nil)
  → (cons a nil) g
  → g a nil False
  → False

Summary

Programming examples

1. an arithmetic expression evaluator (
2. lists as functions (needs thought!)