COMP0020 Functional Programming
Lecture 13
Example Programs

1. Evaluating arithmetic expressions
2. lists as functions

Contents

Programming examples.

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

Programming Example 1

- Writing an evaluator for arithmetic expressions
- 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 simple arithmetic expression using the grammar given on the next page, and
  - returns as output the value of that expression

Grammar (BNF)

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

Preliminaries

- Note that the specification asks for a "representation" of the expression to be input
- Therefore, we can ignore lexical analysis!
- We will also assume that parsing has been done, so we have the syntax tree
- We can choose an algebraic type as our representation of that syntax tree, with appropriate constructors

Algebraic types

expression ::= *Constant num*
| *App expression operator expression*
| *Bracketed expression*

operator ::= *Times* | *Plus* | *Minus* | *Divide*

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) * (3 - 2)
test2 = App
          (Bracketed (App (Constant 4) Plus (Constant 5)))
          Times
          (Bracketed (App (Constant 3) Minus (Constant 2)))
```

Evaluator code

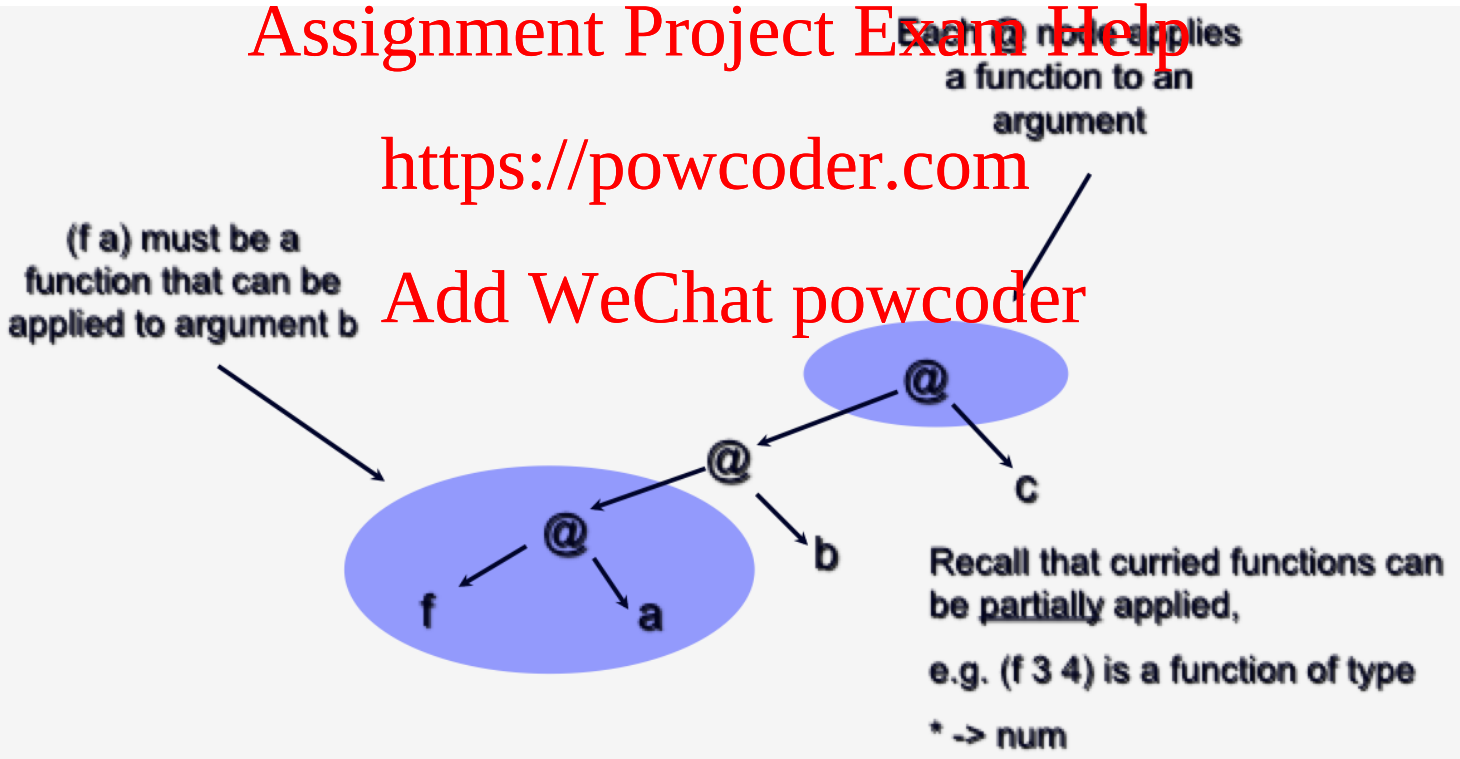
```
eval :: expression -> num
eval (Constant x)       = x
eval (App e1 op e2)     = evalop op (eval e1) (eval e2)
eval (Bracketed e)      = eval e

evalop :: operator -> num -> num -> num
evalop Times x y        = x * y
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 functions and currying
  - Example of how data can be implemented as a function (a "trick" — sometimes useful)

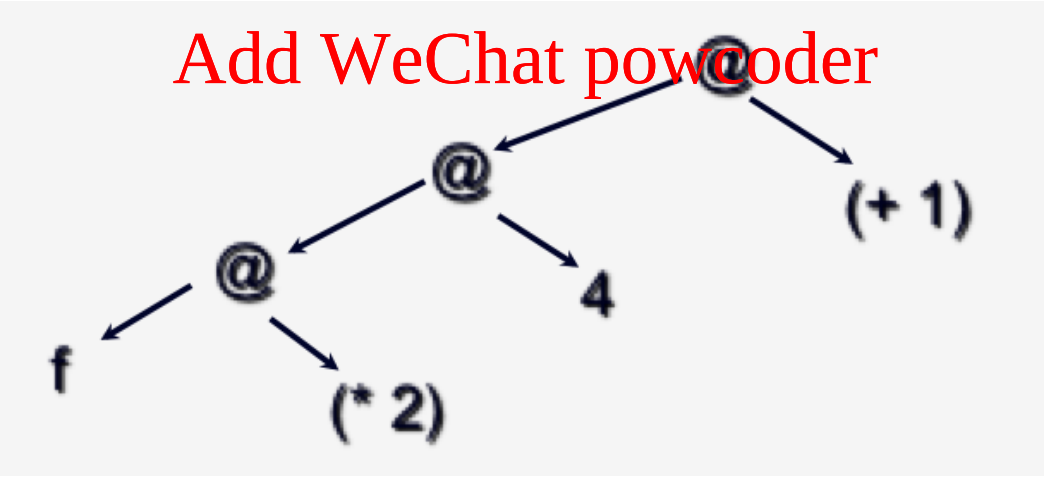
Preamble (1)

- 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 :



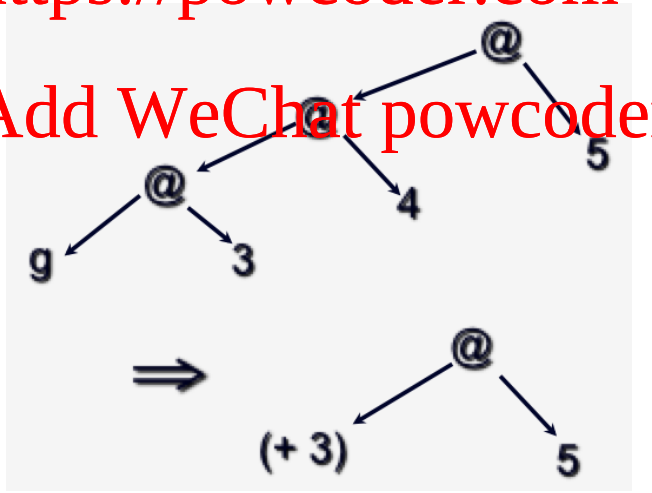
Preamble (2)

- 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 constructors of an algebraic type
  - The difference being that this data structure is now of type **mylist num** and not of type **[num]**
  - This assumes the algebraic type definition :
    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:

$$cons\ a\ b\ f = f\ a\ b\ False$$

$$nil \quad f = f\ (error\ \text{“}head\ of\ nil\text{”})\ (error\ \text{“}tail\ of\ nil\text{”})\ True$$

- 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

			head	tail	isnil
			↓	↓	↓
cons a b f	=	f	a	b	False
nil	=	f	(error "head of nil")	(error "tail of nil")	True


```
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
```

```
||  Example :
||  z = cons 'A' nil
||  y = cons 'B' z
||
||  head y
||  → (cons 'B' z) h
||  → cons 'B' z h
||  → h 'B' z False
||  → 'B'
```

- 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 (simple)
2. lists as functions (needs thought)

END OF LECTURE