

1. [15] Assuming that  $\oplus$  is a left-associative operator having precedence 1 and that  $\otimes$  is right associative and has precedence 2, translate the following infix expression

$$a \oplus b \oplus c \otimes d \otimes e$$

into

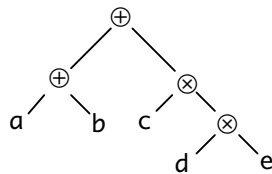
- a. prefix notation

$$\begin{aligned} &(a \oplus b) \oplus (c \otimes (d \otimes e)) \\ &\oplus (\oplus a b) (\otimes c (\otimes d e)) \\ &\oplus \oplus a b \otimes c \otimes d e \end{aligned}$$

- b. postfix notation

$$\begin{aligned} &(a \oplus b) \oplus (c \otimes (d \otimes e)) \\ &(a b \oplus) (c (d e \otimes) \otimes) \oplus \\ &a b \oplus c d e \otimes \otimes \oplus \end{aligned}$$

- c. an abstract syntax tree



2. [15] Given the following grammar:

$\underline{Mars} ::= Venus \mid Mars \& Venus$

$\underline{Venus} ::= Pluto \% Venus \mid Pluto$

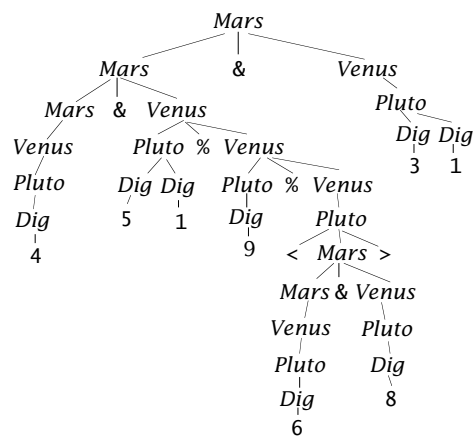
$\underline{Pluto} ::= < Mars > \mid Dig \{ Dig \}$

$\underline{Dig} ::= 0 \mid 1 \mid \dots \mid 9$

- a. [2] If the two operators differ in precedence, circle the one whose precedence is *higher*:  % &
- b. [3] Circle the operator(s) that is (are) right-associative:  % &

- c. [10] Draw the parse tree for this expression:

4 & 51 % 9 % < 6 & 8 > & 31



3. [15] Show the steps in the evaluation of the following Haskell expression

$$(\lambda a b c \rightarrow (a+c)*(a-c)) (2+3) (4*3) (2+1)$$

a. Use an outermost evaluation policy.

$$\begin{aligned} & (\lambda a b c \rightarrow (a+c)*(a-c)) (2+3) (4*3) (2+1) \\ \rightsquigarrow & (\lambda b c \rightarrow ((2+3)+c)*((2+3)-c)) (4*3) (2+1) \\ \rightsquigarrow & (\lambda c \rightarrow ((2+3)+c)*((2+3)-c)) (2+1) \\ \rightsquigarrow & ((2+3)+(2+1))*((2+3)-(2+1)) \\ \rightsquigarrow & (5+(2+1))*((2+3)-(2+1)) \\ \rightsquigarrow & (5+3)*((2+3)-(2+1)) \\ \rightsquigarrow & 8*((2+3)-(2+1)) \\ \rightsquigarrow & 8*(5-(2+1)) \\ \rightsquigarrow & 8*(5-3) \\ \rightsquigarrow & 8*2 \\ \rightsquigarrow & 16 \end{aligned}$$

b. Use an innermost evaluation policy.

$$\begin{aligned} & (\lambda a b c \rightarrow (a+c)*(a-c)) (2+3) (4*3) (2+1) \\ \rightsquigarrow & (\lambda a b c \rightarrow (a+c)*(a-c)) 5 (4*3) (2+1) \\ \rightsquigarrow & (\lambda b c \rightarrow (5+c)*(5-c)) (4*3) (2+1) \\ \rightsquigarrow & (\lambda b c \rightarrow (5+c)*(5-c)) 12 (2+1) \\ \rightsquigarrow & (\lambda c \rightarrow (5+c)*(5-c)) (2+1) \\ \rightsquigarrow & (\lambda c \rightarrow (5+c)*(5-c)) 3 \\ \rightsquigarrow & (5+3)*(5-3) \\ \rightsquigarrow & 8*(5-3) \\ \rightsquigarrow & 8*2 \\ \rightsquigarrow & 16 \end{aligned}$$

c. Use a lazy evaluation policy.

$$\begin{aligned} & (\lambda a b c \rightarrow (a+c)*(a-c)) (2+3) (4*3) (2+1) \\ \rightsquigarrow & (\lambda b c \rightarrow \mathbf{let} a = 2+3 \mathbf{in} (a+c)*(a-c)) (4*3) (2+1) \\ \rightsquigarrow & (\lambda c \rightarrow \mathbf{let} a = 2+3, b = 4*3 \mathbf{in} (a+c)*(a-c)) (2+1) \\ \rightsquigarrow & \mathbf{let} a = 2+3, b = 4*3, c = 2+1 \mathbf{in} (a+c)*(a-c) \\ \rightsquigarrow & \mathbf{let} a = 5, b = 4*3, c = 2+1 \mathbf{in} (a+c)*(a-c) \\ \rightsquigarrow & \mathbf{let} b = 4*3, c = 2+1 \mathbf{in} (5+c)*(5-c) \\ \rightsquigarrow & \mathbf{let} b = 4*3, c = 3 \mathbf{in} (5+c)*(5-c) \\ \rightsquigarrow & \mathbf{let} b = 4*3 \mathbf{in} (5+3)*(5-3) \\ \rightsquigarrow & (5+3)*(5-3) \\ \rightsquigarrow & 8*(5-3) \\ \rightsquigarrow & 8*2 \\ \rightsquigarrow & 16 \end{aligned}$$

4. [15] Define in Haskell a function `quotRem` so that if  $n$  and  $d$  are natural numbers, `quotRem n d` is the pair  $(q, r)$ , where  $q$  and  $r$  are the integer quotient and remainder that result from dividing  $n$  by  $d$ . Your definition may employ addition and subtraction, but it may not employ any multiplication, division, modulo, or remainder functions. Don't neglect to declare the function's type.

a. Use guards.

```
> quotRem :: Int -> Int -> (Int,Int)
> quotRem n d
>   | n >= d    = (1+q,r)
>   | otherwise = (0,n)
> where
>   (q,r) = quotRem (n-d) d
```

An alternative:

```
> quotRem1 :: Int -> Int -> (Int,Int)
> quotRem1 n d = qr 0 n
> where
>   qr :: Int -> Int -> (Int,Int)
>   qr q n
>     | n < d    = (q,n)
>     | otherwise = qr (q+1) (n-d)
```

b. Use only conditional expressions.

```
> quotRem2 :: Int -> Int -> (Int,Int)
> quotRem2 n d = if n >= d then let (q,r) = quotRem2 (n-d) d in (1+q,r) else (0,n)
> quotRem3 :: Int -> Int -> (Int,Int)
> quotRem3 n d = qr 0 n
> where
>   qr q n = if n < d then (q,n) else qr (q+1) (n-d)
```

5. [10] Given three lists

```
as, bs, cs :: [Int]
```

use one or more list comprehensions to define a function `sumProds` so that

$$\text{sumProds as bs cs} = \sum_i a_i \times b_i \times c_i$$

that is, `sumProds` computes the sum of products of corresponding elements of the three lists. You may assume that the lists' lengths are equal or that shorter lists are padded with trailing zeros (which amounts to the same thing). Remember to declare the function's type.

Two solutions:

```
> sumProds :: [Int] -> [Int] -> [Int] -> Int
> sumProds as bs cs = sum [ a*b*c | ((a,b),c) <- zip (zip as bs) cs ]
> sumProds as bs cs = sum [ a*bc | (a,bc) <- zip as [ b*c | (b,c) <- zip bs cs ] ]
```

6. [10] Define in Haskell a function `fromTo` such that

`fromTo a b = [a..b]` for all values `a` and `b` of type `Int`

without using the “`..`” notation. Your answer should include the function’s type.

```
> fromTo :: Int -> Int -> [Int]
> fromTo a b
>   | a>b = []
>   | otherwise = a : fromTo (a+1) b
```

7. [10] Define a function `capWords` which capitalizes only the first letter of each word in a `String`. Assume that words are separated by whitespace characters: space, tab, and new-line. An example:

```
capWords "sgt. pepper's lonely hearts club band" ~ "Sgt. Pepper's Lonely Hearts Club Band"
```

You may use the Prelude function `toUpper :: Char -> Char` which is defined so that `toUpper c` is `c`'s upper-case counterpart if `c` is a lowercase letter, and `c` otherwise. Your function’s type should be included in your answer

A solution:

```
> capWords :: String -> String
> capWords = cap True
> where
> cap :: Bool -> String -> String
> cap _ [] = []
> cap b (c:cs) = (if b then toUpper c else c) : cap (c `elem` " \n\t") cs
```