

1. [12] Translate the following postfix expression

$a\ b\ c\ *\ +\ d\ e\ \&\ \#\ f\ \wedge$

assuming that letters are operands, all other characters are operators, and every operator takes two arguments,

- a. to prefix notation.

$a\ b\ c\ *\ +\ d\ e\ \&\ \#\ f\ \wedge$

$((a\ (b\ c\ *)\ +)\ (d\ e\ \&)\ \#)\ f\ \wedge$ -- add parentheses

$\wedge\ (\#\ (+\ a\ (*\ b\ c))\ (\&\ d\ e))\ f$ -- move operators

$\wedge\ \#\ +\ a\ *\ b\ c\ \&\ d\ e\ f$ -- remove parentheses

- b. to infix notation, assuming that the operators have the following associativities and precedences:

$*$ left, 2

$+$ left, 2

$\&$ right, 1

$\#$ right, 1

\wedge left, 3

Include parentheses only where necessary.

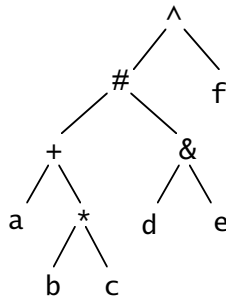
$a\ b\ c\ *\ +\ d\ e\ \&\ \#\ f\ \wedge$

$((a\ (b\ c\ *)\ +)\ (d\ e\ \&)\ \#)\ f\ \wedge$ -- add parentheses

$((a\ +\ (b\ *\ c))\ \#\ (d\ \&\ e))\ \wedge\ f$ -- move operators

$(a\ +\ (b\ *\ c)\ \#\ d\ \&\ e)\ \wedge\ f$ -- remove parentheses

- c. to an abstract syntax tree.



2. [10] Modify the following expression grammar

$\langle pak \rangle ::= \langle pak \rangle \$ \langle maple \rangle | \langle maple \rangle$

$\langle maple \rangle ::= a | b \dots | z$

to allow a right-associative operator (\bullet) which has lower precedence than ($\$$).

A solution:

$\langle pak \rangle ::= \langle birch \rangle \bullet \langle pak \rangle | \langle birch \rangle$

$\langle birch \rangle ::= \langle birch \rangle \$ \langle maple \rangle | \langle maple \rangle$

$\langle maple \rangle ::= a | b \dots | z$

3. [10] Given this grammar

Spruce ::= *Spruce* & *Hemlock* | *Hemlock*

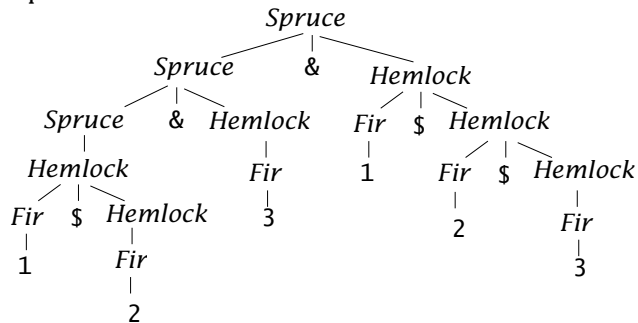
Hemlock ::= *Fir* | *Fir* \$ *Hemlock*

Fir ::= 1 | 2 | 3 | 4 | 5 | 6

draw a parse tree for the expression

1 \$ 2 & 3 & 1 \$ 2 \$ 3

The parse tree:



4. [15] Show the steps in the evaluation of the expression

f (3<0) (3+4+5) (6+7+8)

where f is defined by

f x y z = if x then y*y else z*z

Show each operation in a separate step— don't combine two operations in a single step.

a. Use leftmost innermost evaluation.

f (3<0) (3+4+5) (6+7+8)
 ~ f false (3+4+5) (6+7+8)
 ~ f false (7+5) (6+7+8)
 ~ f false 12 (6+7+8)
 ~ f false 12 (13+8)
 ~ f false 12 21
 ~ if false then 12*12 else 21*21
 ~ 21*21
 ~ 441

b. Use leftmost outermost non-lazy evaluation.

f (3<0) (3+4+5) (6+7+8)
 ~ if (3<0) then (3+4+5)*(3+4+5) else (6+7+8)*(6+7+8)
 ~ if false then (3+4+5)*(3+4+5) else (6+7+8)*(6+7+8)
 ~ (6+7+8)*(6+7+8)
 ~ (13+8)*(6+7+8)
 ~ 21*(6+7+8)
 ~ 21*(13+8)
 ~ 21*21
 ~ 441

c. Use leftmost lazy evaluation.

```
f (3<0) (3+4+5) (6+7+8)
~> let x = 3<0, y = 3+4+5, z = 6+7+8 in if x then y*y else z*z
~> let x = false, y = 3+4+5, z = 6+7+8 in if x then y*y else z*z
~> let x = false, y = 3+4+5, z = 6+7+8 in if false then y*y else z*z
~> let z = 6+7+8 in z*z
~> let z = 13+8 in z*z
~> let z = 21 in z*z
~> 21*21
~> 441
```

5. [18] Rewrite the following Haskell function definition

```
> take :: Int -> [a] -> [a]
> take 0 _          = []
> take _ []         = []
> take n (x:xs) | n>0 = x : take (n-1) xs
> take _ _         = error "take: negative argument"
```

in three different ways, preserving the order of the tests:

a. [10] using guards only (i.e., with no patterns)

```
> take n xs
> | n == 0    = []
> | null xs  = []
> | n > 0    = head xs : take (n-1) (tail xs)
> | otherwise = error "take: negative argument"
```

b. [5] using conditional expressions only (i.e., with no guards or patterns)

```
> take n xs = if n == 0 then []
>             else if null xs then []
>             else if n > 0 then head xs : take (n-1) (tail xs)
>             else error "take: negative argument"
```

c. [3] as a lambda expression, using conditional expressions only

```
> take = \ n xs -> if n == 0 then []
>             else if null xs then []
>             else if n > 0 then head xs : take (n-1) (tail xs)
>             else error "take: negative argument"
```

6. [10] Define in Haskell a function `cJustify` so that `cJustify n s` is a `String` of length `n` containing `s`, preceded and followed by `String`s of spaces which differ from each other by at most 1 (if `s`'s length exceeds `n`, then `cJustify n s ~> s`). Examples (spaces denoted by `□` for visibility:

```
cJustify 10 "Jester" ~> "□□ Jester□□"
cJustify 11 "Jester" ~> "□□ Jester□□□"
cJustify 0 "Jester"  ~> "Jester"
```

Make sure to give `cJustify`'s type.

A solution:

```
> cJustify :: Int -> String -> String
> cJustify n s = spaces before ++ s ++ spaces after
>   where
>     before = (n - lenS) `div` 2
>     after  = n - lenS - before
>     lenS   = length s
>     spaces n = replicate n ' '
```

7. [10] Define in Haskell a function `transp` so that assuming `xss` is a non-empty list of non-empty lists representing a matrix, then `transp xss` is a list of lists representing the transpose of the matrix. For example,

```
transp [[0,1,2],[3,4,5],[6,7,8],[9,10,11]] ~=[0,3,6,9],[1,4,7,10],[2,5,8,11]
```

Do *not* make your solution explicitly recursive— use one or more list comprehensions. And don't concern yourself with efficiency; what's important is your solution's clarity and correctness.

A solution:

```
> transp :: [[a]] -> [[a]]  
> transp xss = [ [ xs!!n | xs <- xss ] | n <- [0 .. length (head xss) - 1 ] ]
```