

Reading

19. Wednesday 2 Mar: Sethi, Sections 3.5, 3.6.
 20. Friday 4 Mar: Sethi, Sections 3.4, 3.7
 21. Monday 7 Mar: Sethi, Section 3.2
 22. Wednesday 9 Mar: Sethi, Sections 3.3, 4.1, 4.9

Exercises

Due date: Tuesday 8 March. Please leave your paper in the Taylor Hall homework box (it's located between 2.132 and 2.136) **by 4pm.**

Several of these exercises require you to write Haskell programs. You should run each program on Hugs, and include some output, demonstrating the function, immediately following your Haskell code.

1. [30] The expression evaluator presented in class is rather crude—its parser assigns no precedence or associativity to operators. As a step towards a better parser, define
- a. a parser `leftAssocSub` which parses and evaluates expressions whose syntax is

$LeftAssocSub ::= LeftAssocSub - Num \mid Num$

Make sure that `leftAssocSub` handles `(-)` operators *left*-associatively; for example,

`papply leftAssocSub "3-2-1" ~> [(0,"")]` **not** `[(2,"")]`

As noted in class, a straightforward implementation such as

```
> leftAssocSub = do n1 <- leftAssocSub
>                  char '-'
>                  n2 <- num
>                  return (n1 - n2)
>
>                  +++
>                  num
```

is doomed—whenever `leftAssocSub` is applied to a string, it begins by applying itself to exactly the same string, resulting immediately in nonterminating recursion.

An alternative definition is suggested by an alternative formulation of the grammar:

$LeftAssocSub ::= Num \{ - Num \}$

This can be implemented successfully by

```
> leftAssocSub = do n <- num; nums n
```

where `nums`, which is defined recursively, implements $\{ - Num \}$.

- b. a parser `rightAssocPow` which parses and evaluates expressions whose syntax is

$RightAssocPow ::= Num \wedge RightAssocPow \mid Num$

Make sure that `rightAssocPow` handles `(^)` operators *right*-associatively; for example,

`papply rightAssocPow "2^3^2" ~> [(512,"")]` **not** `[(64,"")]`

The right-recursive production can be implemented successfully by a recursive definition of `rightAssocPow`.

2. [15] As the next step in the improvement of the expression evaluator, define an expression parser and evaluator, `expr`, which handles this syntax:

```
Expr ::= Term { ( + | - ) Term }
Term ::= Fac { ( * | / ) Fac }
Fac  ::= Pow ^ Fac | Pow
Pow  ::= '(' Expr ')' | Num
Num  ::= Digit { Digit }
Digit ::= 0 | 1 | ... | 9
```

This resembles the grammar of Slide 29, but we've added a right-associative exponentiation operator `(^)`.

Demonstrate your parser/evaluator, using `papply`, on some examples which show that the operators have the correct associativity and precedence.

3. [30] Install your solution for Exercise 2 in a copy of the interactive expression evaluator shown in Slides 122–124. This code is available on-line at http://www.cs.utexas.edu/users/ham/UTCS/CS345/Hugs_scripts/Evaluator.lhs. To improve expressions' readability, add provisions to allow zero or more spaces on either side of operators, ":", '=', '(', and ')'.
Finally, arrange for your evaluator to distinguish between syntax errors and attempts to divide by zero. A simple way to handle the latter is to define a type synonym—say, `Number`—which pairs each `Int` with a `Bool` indicating whether the number is valid.

4. [15] Addition of natural numbers is of course built-in to Haskell. It's possible, however, to define it recursively, in terms of the more rudimentary successor function (`Succ`) and constant `Zero`:

$$\begin{aligned} m + \text{Zero} &= m \\ m + (\text{Succ } n) &= \text{Succ } (m + n) \end{aligned}$$

(`Succ` is a special kind of function—a constructor function like `(:)`—and like `(:)` it can be used in patterns.)

Using this definition it is possible to prove the following theorems, which are familiar properties of natural-number arithmetic:

- | | | |
|----|-----------------------------|--------------------------------|
| 1. | $\text{Zero} + n = n$ | Zero is the left unit of $(+)$ |
| 2. | $a + (b + c) = (a + b) + c$ | $(+)$ is associative |
| 3. | $m + n = n + m$ | $(+)$ is commutative |

For this exercise, prove theorems 2 and 3. In each of your proofs, you may use *only*

- the definition of $(+)$ given above
- any theorem with a smaller theorem number.

In proving Theorem 3, you may find it useful to construct an auxiliary theorem, i.e., a lemma.

5. [5] Prove that for all finite lists `xs`,
- $$xs ++ [] = xs$$

The definition of `(++)`, for reference:

$$> [] ++ bs = bs \quad (1)$$

$$> (a:as) ++ bs = a : (as++bs) \quad (2)$$

6. [15] Prove formally that `map` can be defined nonrecursively using `foldr`. Specifically, prove that for all lists `xs`,

$$\text{map } f \text{ } xs = \text{foldr } ((:).f) [] \text{ } xs$$

The definitions, for reference:

$$> \text{map } f [] = []$$

$$> \text{map } f (a:as) = f a : \text{map } f as$$

$$> \text{foldr } f k [] = k$$

$$> \text{foldr } f k (b:bs) = f b (\text{foldr } f k bs)$$

$$> (g.h) z = g (h z)$$

CS 345 Richards

Assignment 5

Authors:

1. _____

2. _____

3. _____

4. _____