INFOMCPD Assignment 0: Getting started with Haskell

Tom Smeding, Gabriele Keller

November 15, 2022

1 Installing the Haskell compiler

To do the assignments it is most convenient to be able to develop on your own computer. The best way to install GHC, the primary Haskell compiler, is to use GHCup: go to `https://www.haskell.org/ghcup/` and follow the instructions for your platform. During the GHCup installation, be sure to install GHC and Cabal, but not Stack[1]; if you want interactive compiler feedback in your editor, also install HLS[2]. (For VSCode there is a Haskell plugin that works well with HLS.)

If the GHCup installation does not work, first ask us (we might know what is going wrong and be able to fix it). If we cannot fix it, an alternative is to install Stack using the instructions at `https://docs.haskellstack.org/en/stable/`; if you do this, you will need the `stack` instructions instead of the `cabal` instructions in the rest of this document.

On Windows, a common cause for GHCup installation failure is overly aggressive anti-virus software; this may be the cause if the initial part of the script seems to work but downloading the actual software later fails. If you run into this, a fix is probably to remove your current anti-virus and install Microsoft Defender instead.

Haskell tutorial: the rest of this assignment will assume that you have some basic knowledge of how Haskell works, or at least that you know how to look things up. A good tutorial is `http://learn.hfm.io/`; note that there is a small amount of overlap with the end of that tutorial and this assignment.

2 Submission and grading

We do not yet have the submission system up for this assignment; we will update this assignment specification (and notify you) when this changes. Submission instructions will be up well before the deadline of the first part of this assignment (The RPN calculator, Section 4) is due.

This assignment has three parts: 0.1, 0.2 and 0.3. All three parts are mandatory:

- Assignment 0.1 (Section 4): 3% of your final grade (0.3 grade point) (due 2022-12-02 23:59)
- Assignment 0.2 (Section 5): 3% of your final grade (0.3 grade point) (due 2022-12-09 23:59)
- Assignment 0.3 (Section 6): 4% of your final grade (0.4 grade point) (due 2022-12-16 23:59)

All together, assignment 0 (this document) is 10% of your final grade, giving you a 1.0 grade if you would do nothing else in the course.

[1] While installing Stack is not harmful at all, Cabal and Stack have roughly the same purpose and having both can be confusing at times, especially with language-server-based editor support.

[2] Haskell Language Server

You get the points for a part if you submit a working program that conforms to the specification. Programming style is not graded, though you can of course ask for feedback on style and approach during the lab sessions.

Why? This assignment is intended to let you get somewhat familiar with Haskell in a simple setting, so that the larger assignment later in the course will allow you to focus more on the content and less on learning the language. We made it mandatory on purpose: if you do not know Haskell well yet, it is a good exercise or a good refresher; if you are already adept at Haskell, it should take you very little time. Hence, in neiter case do you have an excuse. :)

3 Reverse Polish Notation (RPN)

Since the first assignment (Section 4) is about arithmetic expressions in postfix notation, also called Reverse Polish Notation, here is a brief introduction about how those work. Normal arithmetic expressions have operators like `+` and `*` *between* their arguments; this is called *infix* notation. In infix, the sum of 1 and 2 is written `1 + 2`, and the product of that and 3 is written `(1 + 2) * 3`. Note that we needed to use parentheses to disambiguate the order of operations; without parentheses and without precedence rules, it would be ambiguous whether `1 + 2 * 3` meant `(1 + 2) * 3` or `1 + (2 * 3)`.

In postfix notation, the operators are written *after* their arguments. For example, the sum of 1 and 2 is written `1 2 +`, and the product of that and 3 is written `1 2 + 3 *`. Note that we do not need to use parentheses here; indeed, looking from the back, the multiplication `*` needs two arguments. The second argument is clearly the 3, and the first argument must then be the next previous thing — the addition of 1 and 2.

An alternative interpretation of postfix notation is by evaluation using a stack. For example, evaluation of the expression `2 3 7 + 4 * +` would proceed as follows:

Initial stack:			
After `2`:	2		
After `3`:	2	3	
After `7`:	2	3	7
After `+`:	2	10	
After `4`:	2	10	4
After `*`:	2	40	
After `+`:	42		

You will not need to write a stack-based evaluator in this style, but we hope this helps to clarify the meaning of a postfix expression.

4 Assignment 0.1: RPN

You are going to write an interpreter for postfix (RPN) expressions. You will only have to support non-negative integers, and only addition (`+`) and multiplication (`*`). In addition to the examples given in Section 3, here are some more RPN expressions that you will need to be able to handle:

Input expression	**Evaluation**	
`12 3 * 6 +`	$(12 \cdot 3) + 6 = 42$	
`1 2 3 4 5 + + + +`	$1 + (2 + (3 + (4 + 5))) = 15$	
`123 0 + 2 2 * +`	$(123 + 0) + (2 \cdot 2) = 127$	
` 123 0+2    2*   + `	$(123 + 0) + (2 \cdot 2) = 127$	— *You need to handle this too*

Your program will be split in three parts:

1. You will write one function to split the input into tokens: numbers, plus signs and multiplication signs.
2. Then you will write a function to parse the input expression (in *concrete syntax*) to an internal representation (the *abstract syntax*).
3. Finally, you will write a function that evaluates the expression in this internal, abstract representation to its integer result.

Due to the simplicity of this task, going through the intermediate representation might feel redundant; please follow this two-part structure anyway, because it allows you get practice with some relevant concepts in this course.

4.1 Setting up your development environment

Create a file called, for example, `rpn.hs`. Open it in your editor, and to the side of your editor, open a terminal (or `cmd` on Windows) in the directory where your file is stored. Then type `ghci rpn.hs` in the terminal (if you installed via Stack instead of GHCup, type `stack repl rpn.hs`). Make sure that opens correctly and asks for input with a prompt that looks like `ghci>` or `*Main>`.

Next, put `foo = "hello!"` your file, and type `:r` (followed by Enter) in your terminal (i.e. in ghci). It should reload the module, and afterwards you should be able to type `foo` (and Enter), which will show `"hello!"`. You can remove the definition of `foo` from the file now that you have verified that ghci works.

Recommendation: Put the following line all the way at the top of your program:

```
{-# OPTIONS -Wall #-}
```

This enables warnings in ghci, such as when your pattern matches are not exhaustive (i.e. when they do not cover all possibilities). This is very helpful while developing.

4.2 Lexing

Note: There is a template file for this assignment here: https://www.cs.uu.nl/docs/vakken/mcpd/2021/website/exercises/asg-0-1-template.hs

Important: *Even if you don't use the template, you must start your file with* `module RPN where`*. This module name is necessary for grading to proceed correctly.*

Splitting up the input string into tokens is often called *tokenisation* or *lexing*. Define a data type called `Token` that has three constructors: one for numbers (with an `Int` field), one for plus signs and one for multiplication signs. The second and third constructor of your `Token` type do not need any fields; this is almost like an enumeration type.

Don't forget to make your `Token` type showable! You do this by appending `deriving (Show)` after your data type definition, like so:

```
data Token = ...
  deriving (Show)
```

This ensures that you will have a *much* nicer debugging experience in ghci.

After you have defined `Token`, define a function called `lexer` with the type signature `String -> [Token]` that will take the input string and produce a list of tokens. If some invalid input is detected, the function is allowed to error out using `error`; you do not need to do proper error handling in this assignment.

Be sure to use pattern matching to walk over the input, and recursion to parse “the rest” of the input after you have parsed one token. Functions that may come in handy:

- `isDigit :: Char -> Bool` [3] (you need to import `Data.Char` for this)
- `read :: String -> Int` [4]; actually this function has a much more general type, but you can use it at this type as well. If you want to try `read` in ghci, write something like `read "123" :: Int` to constrain its return type to `Int`.
- `span :: (a -> Bool) -> [a] -> ([a], [a])` [5]: this function returns the prefix of the given list that satisfies the predicate, as well as the rest of the list.

You can test your function in ghci (remember to type `:r` first to reload the file): for example, try entering `lexer "1 2 +"` in ghci, or see what error is thrown on invalid input such as in `lexer "1!"` or `lexer "abc"`.

```
ghci> lexer "1 2 +"
[TNum 1,TNum 2,TPlus]
ghci>
```

(You do not need to name the constructors of your `Token` data type like I did — though these are common choices.)

4.3 Parsing

After you have split the input into tokens, you will now parse them into an expression tree. Define a data type called `Expr` with three constructors: one for numbers again, and two for addition and multiplication, each with two fields of type `Expr`. This type is an abstract representation of an arithmetic expression: there is no concept of parentheses or any kind of syntax anymore. Furthermore, it is unambiguous: there is no ambiguity in the order of operations like with `(1 + 2) * 3` versus `1 + (2 * 3)`.

The actual parsing will be done by a helper function `parser' :: [Expr] -> [Token] -> Expr`, where the `[Expr]` argument is a *stack* of expressions, and the `[Token]` argument is the list of tokens still to parse. The parsing model is similar to the evaluation example in Section 3, except here the stack contains expressions instead of integers. The top-level parsing function, `parser :: [Token] -> Expr`, will simply initialise the stack with the empty stack, `[]`, and delegate the rest to `parser'`.

In `parser'`, recurse over the input token list, pattern matching on (the top of) the stack whenever you need some expressions from it (i.e. when you encounter the tokens for plus signs and multiplication signs). You can push something on the stack simply by passing the enlarged stack to the recursive call to `parser'`.

Again, test your function (or some half-done version of your function) in ghci by running it on some inputs (and see what error is thrown on invalid input such as `parser (lexer "1 +")` or `parser (lexer "1 1")`). You can either go through your lexer and run `parser (lexer "1 2 +")` or you can pass a list of tokens directly: the list notation that you get from running `lexer "1 2 +"` is valid Haskell syntax and can be typed in directly as an argument to `parser`.

[3] https://hackage.haskell.org/package/base-4.16.3.0/docs/Data-Char.html#v:isDigit
[4] https://hackage.haskell.org/package/base-4.16.3.0/docs/Prelude.html#v:read
[5] https://hackage.haskell.org/package/base-4.16.3.0/docs/Prelude.html#v:span

4.4 Evaluation

After you have written the parser, all that is left is to evaluate the arithmetic expression in Expr. For this, write a recursive eval function of type Expr -> Int. This function will be quite short.[6]

A note: in a way, this eval function follows the *big-step semantics* of arithmetic expressions. What exactly this means you will learn later on in the course.

After this, you have two data types (Token and Expr) and three functions (lexer, parser and eval). (Actually, you also have parser', but that is simply a helper function for parser.) With that, this task is done.

5 Assignment 0.2: Infix

You received complaints from your users that they want to write expressions in infix form, not in prefix form. So you get to work.

Note: There is a template file for this assignment here: https://www.cs.uu.nl/docs/vakken/mcpd/2021/website/exercises/asg-0-2-template.hs

Important: Start your file with module Infix where — this line is necessary for grading to proceed correctly.

You will update the lexer and the parser to now parse infix expressions with non-negative numbers, addition, multiplication and parentheses. For this, your Token type will need to be expanded as well. However, the expression returned from your new parser will be expressed in the same Expr data type as in Assignment 0.1; indeed, prefix versus infix is just a choice of *concrete syntax*. The *abstract syntax*, here the Expr data type, does not change, and neither does the *semantics*, your evaluator eval.

Name your updated token type TokenInfix, and name your updated lexer and parser functions as follows:

```
lexerInfix :: String -> [TokenInfix]
parserInfix :: [TokenInfix] -> Expr
```

You *do* need to handle operator precedence (order of operations): 1 * 2 + 3 should parse the same as (1 * 2) + 3, and 1 + 2 * 3 should parse the same as 1 + (2 * 3). However, the operators do not need to be left-associative: you can parse 1 + 2 + 3 + 4 either as ((1 + 2) + 3) + 4 or as 1 + (2 + (3 + 4)); both are allowed.

Some examples that you can test your code on:

Input expression	**Result**
3 + 12	15
3*12	36
3 * (12 + 2)	42
(3 * 12) + 2	38
3 * 12 + 2	38
12 + 3 * 6	30
1 + 2 + 3 + 4 + 5	15
1 + 2+ 3 *4+5	20
(1 + (((2) * 7)) + 3)*(4+5)	162

Your previous lexer and parser functions are allowed to remain in your submission for Assignment 0.2, but they don't have to.

[6]For the mathematicians in the room: eval is a nice example of a catamorphism.

6 Assignment 0.3: Lambda

Your users are again complaining; they wanted a programming language, not a calculator. So you get to work again. This time, you will be writing a very simple interpreter for the lambda calculus with numbers, addition and multiplication — but as before, you are going to give your users exactly what they asked for, and nothing more. Indeed, your users did not ask for an *easy to use* programming language.

This assignment is separate from the last two; it will be a single file again, but you should start from scratch.

Note: There is a template file for this assignment here: `https://www.cs.uu.nl/docs/vakken/mcpd/2021/website/exercises/asg-0-3-template.hs`

Important: Start your file with `module Lambda where` — this line is necessary for grading to proceed correctly.

The syntax of your language is as follows:

$$\frac{v\ \text{Var} \quad e\ \text{Expr}}{\#ve\ \text{Expr}}\ \textsc{Lambda} \qquad \frac{e_1\ \text{Expr} \quad e_2\ \text{Expr}}{`e_1 e_2\ \text{Expr}}\ \textsc{App} \qquad \frac{v\ \text{Var}}{v\ \text{Expr}}\ \textsc{Var} \qquad \frac{n\ \text{Num}}{n\ \text{Expr}}\ \textsc{Num}$$

where the Var judgement matches on a sequence of 1 or more letters (a–z A–Z) and the Num judgement matches on a sequence of 1 or more digits (0-9).

For example, the following example programs have the listed meaning in normal lambda calculus notation:

Input expression	**Meaning in normal notation**
` f x	$f\ x$
` ` f x y	$f\ x\ y$
` f ` g x	$f\ (g\ x)$
# x ` ` add x 1	$\lambda x.$ add x 1
#x``add x 1	$\lambda x.$ add x 1
` #x ``add x 1 10	$(\lambda x.$ add x 1) 10
` f #x ``add x 1	$f\ (\lambda x.$ add x 1)
` #f `f 10 #x ``mul 2 x	$(\lambda f.\ f$ 10) $(\lambda x.$ mul 2 $x)$
` #inc `inc 10 #x ``add 1 x	$(\lambda inc.\ inc$ 10) $(\lambda x.$ add 1 $x)$

And some more examples, all of which evaluate to an integer:

Input expression	**Result**
`` add 2 3	5
`` mul 11 3	33
`` mul 6 `` add 3 4	42
` #inc `inc 10 #x ``add 1 x	11
``#r#t```r t`add 1 0#f```f f f f#f#x`f`f x	65536

Split your implementation into three parts: a lexer (that splits up the input `String` into a list of tokens, `[Token]`), a parser (that parses a list of tokens `[Token]` into an expression `Expr`), and an evaluator (that evaluates an `Expr` to a `Value`). Note that because not all expressions evaluate to integers any more in this language, the evaluator cannot simply return an `Int`.

Instead, define a data type `Value` with two constructors, one for integers and one for functions. This data type looks as follows:

```
data Value = VInt Int | VFun (Value -> Value)
  deriving (Show)
```

You must use this data type — though you are allowed to change the constructor names `VInt` and `VFun` if you want.

You will notice that if you put `deriving (Show)` on this definition, Haskell will reject your code, saying that there is no instance of `Show` defined for `Value -> Value`. Indeed, by default, Haskell defines no method for printing functions, as that is kind of hard to do in general. However, you can cop-out by writing:

```
import Text.Show.Functions ()
```

which adds a dummy instance that shows any function value as `<function>`. With that, `deriving (Show)` on `Value` should be accepted.

6.1 The evaluator

Since this language has support for variables, your evaluator will need to get a slightly different type. Import the following modules:

```
import Data.Map.Strict (Map)
import qualified Data.Map.Strict as Map
```

This allows you to use the `Map` type in your code, with the `Map` operations written as for example `Map.insert`. The documentation for this type in the sported operations can be found at https://hackage.haskell.org/package/containers-0.6.5.1/docs/Data-Map-Strict.html. Particularly relevant operations are `empty`, `insert` and `lookup`[7].

A value of type `Map k v` is a finite map from keys of type `k` to values of type `v` — also known as dictionaries, tree-maps, hashmaps (although in this case they are in fact tree-maps), etc. You will store the variables currently in scope, together with their values, in a `Map` in your evaluator.

Define the core of your evaluator with this type:

```
eval' :: Map String Value -> Expr -> Value
```

and then define a (very short) wrapper that initialises evaluation with an empty environment:

```
eval :: Expr -> Value
```

The top-level interpreter. Finally, define the following function:

```
interpret :: String -> Value
```

that composes your lexer, your parser and your evaluator. The `interpret` function is the function that will be evaluated when the grading your submission for Assignment 0.3. Again you may assume that all input is valid, i.e. you may throw errors using `error` if you do encounter invalid input.

[7] Consume the result of `Map.lookup` using a `case` expression.