

Advanced Functional Programming

Universität Tübingen

Weekly exercise sheet 4

Instructions The answers to this sheet have to be submitted at <https://cis.informatik.uni-tuebingen.de/afp-ws-1112/> before 14-11-11 14:00.

You are allowed to work in teams of 2.

The answers to these exercises have to be submitted in a Haskell script file.

The file should load into ghci without any errors. *Failing to do so will cost you 10 points.*

At the top of the file state your name in a comment.

Mark the answer to each question with a Haskell comment (`-- exercise 1`).

Non code answers have to be answered in a Haskell comment. If your answer is a single line answer you can use a single line comment (`-- your answer`), if you need more space enclose your answer in a multiline comment (`{- your answer -}`).

Be sure to add a type signature to **all** function declarations (that is top level functions and values).

Read carefully For this weeks set of exercises you'll be working with the Parser library we defined during the lecture. You can download the parser library from: <http://db.inf.uni-tuebingen.de/files/weijers/AFP1112/ParseLib.hs>. This module exposes all the combinators defined during the lecture and a few more. You have to import this module into your solutions file. You are *not* allowed to modify this file in any way. It doesn't expose the constructor P of the parser, meaning you cannot define any primitives (but you can combine other parsers).

Exercise 1 (15 points)

a.) In the lecture we defined a grammar containing two operators addition and multiplication and we parsed them in a right associative way. That means if we encounter a string $x1 \oplus x2 \oplus x3 \oplus x4$ we parse it as if there were parentheses as follows: $x1 \oplus (x2 \oplus (x3 \oplus x4))$. Define an operator *chainr* that parses as many elements as possible (but at least one) separated by a right associative operator. The type signature of this parser has to be:

$$\text{chainr} :: \text{Parser } a \rightarrow \text{Parser } (a \rightarrow a \rightarrow a) \rightarrow \text{Parser } a$$

The first argument should parse the elements, the second argument should parse the operator and return a function that combines two elements separated by this element. The combination function should be applied in such a way that the elements are combined in a right associative manner.

b.) The same as in exercise 1.a, but now define *chainl* for left associative operators (this function has the same type).

c.) Define a combinator *sepBy* that given two parsers parses zero or more occurrences of the elements recognised by the first parser separated by elements recognised by the second parser. The function returns a parser of a list of elements recognised by the first parser. The type is:

$$\text{sepBy} :: \text{Parser } a \rightarrow \text{Parser } b \rightarrow \text{Parser } [a]$$

Exercise 2 (10 points)

Define a parser *binPal* :: *Parser String* for the language of binary palindromes. A palindrome is a word that reads the same both backwards and forwards. The language of binary palindromes is described by the following grammar:

$$\text{Pal} ::= "0" \text{ Pal } "0" \mid "1" \text{ Pal } "1" \mid "1" \mid "0" \mid \epsilon$$

Make sure the parser parses the whole string (and fails if the string isn't a palindrome). Hint: the parser *eof* may turn out to be useful in this exercise.

Exercise 3 (15 points)

Consider the following grammar:

```

Expr ::= "\" Var "->" Expr
      | "if" Expr "then" Expr "else" Expr
      | Expra Expra
      | Expra
Expra ::= Term ("+" Expr | ε)
Term  ::= Factor ("*" Term | ε)
Factor ::= Val | "(" Expr ")"
Val    ::= Bool | Nat | Var
Bool   ::= "True" | "False"
Nat    ::= Digit Nat | Digit
Digit  ::= "0" | ... | "9"

```

The productions for *Var* can be parsed by the parser *identifier*. Extend this language with support for tuples of arbitrary size (tuples occur at the level of factors and contain arbitrary expressions) and a selection operator "@". The selection operator selects the *n*th value from a tuple. It binds stronger than multiplication. On the left hand side it should expect a *factor* on the right hand side a natural number. This operator does not have to be associative. Parentheses may be required to select a tuple in a tuple: $(e@1)@2$, selects the second component of the first component of e . If you want to use the operator multiple times without parentheses make sure it is left associative. You also have to expand the data type and the evaluator function.

A parser and the evaluator for this language can be download here <http://db.inf.uni-tuebingen.de/files/weijers/AFP1112/Parsing.hs>, copy and paste this code into your solution file.

Exercise 4 (master/diploma students, 5 points)

a.) Define a parser *palAZ* :: *Parser String* that recognises palindromes written using the letters 'a' ... 'z' (your parser can fail on uppercase letters). You are *not* allowed to write all the production rules explicitly (which is a laborious task). Hint: write a function that generates the production rules.

Optional (5 bonus points)

b.) Change the parser from Exercise 3 so that it can also parse subtraction (same binding strength as addition) and division (same binding strength as multiplication). These two operators are left associative, you can use the *chainl* combinator from Exercise 1 to achieve this. You have to change the associativity of addition and multiplication into left associativity (which way they associate doesn't matter for these operators, in Haskell they are also left associative).