Appendix E

FP Project

In the FP project we revisit most topics from the exercises. Please read this *entire* document before you start programming.

The project consists of a few parts wherein you create (some of) the following using Haskell:

- Your own parser combinator library (Section E-3)
- Basic parsers defined using your parser combinators (Section E-4)
- An embedded domain specific language (EDSL) with deep embedding for a simple functional language called *μFP*, or "microFP" (Section E-5)
- An evaluator for your EDSL (Section E-5)
- A parser for a grammar for μFP defined using your parser combinators (Section E-6)
- Some further features, including test code (Section E-7)

E-1 μFP

Within the project you develop a parser and evaluator for a language called μFP. μFP is a (pure) functional language with the following grammar:

⟨*program*⟩ ::= (⟨*function*⟩)+
⟨*function*⟩ ::= `identifier` (`identifier` | `integer`)* `':='` ⟨*expr*⟩ `';'`
⟨*expr*⟩ ::= ⟨*term*⟩ | ⟨*term*⟩ (`'+'` | `'-'`) ⟨*expr*⟩
⟨*term*⟩ ::= ⟨*factor*⟩ | ⟨*factor*⟩ `'*'` ⟨*term*⟩
⟨*factor*⟩ ::= `integer`
| `identifier` (`'('` ⟨*expr*⟩ (`','`⟨*expr*⟩)*`')'`)?
| `'if' '('` ⟨*expr*⟩ ⟨*ordering*⟩ ⟨*expr*⟩ `')' 'then' '{'` ⟨*expr*⟩ `'}' 'else' '{'` ⟨*expr*⟩ `'}'`
| `'('` ⟨*expr*⟩ `')'`
⟨*ordering*⟩ ::= `'<'` | `'=='` | `'>'`

The identifiers in this language start with a lower case letter, followed by zero or more numbers and/or lower case letters. Features of the language include multiple arguments, pattern matching, higher order functions and partial function application. Some important differences with respect to Haskell are:

- Function definitions *must* end with a semicolon.
- Function application is *not* a space; instead the arguments are supplied between parentheses as comma separated list. Note, that this may look a bit funny when partial application is used.
- If-expressions use parentheses for the Boolean expression, and braces around the two alternative expressions.
- No built-in division.

Furthermore, since this project is about Functional Programming and not about Compiler Construction, associativity and operator precedence are not important.

Below are some valid functions for μFP:

```
fibonacci 0 := 0;
fibonacci 1 := 1;
fibonacci n := fibonacci (n-1) + fibonacci (n-2);

fib n := if (n < 3) then {
           1
     } else {
       fib (n-1) + fib (n-2)
     };

sum 0 := 0;
sum a := sum (a-1) + a;

div x y :=
   if (x < y) then
     {
       0
     } else {
       1 + div ((x-y), y)
     };

twice f x := f (f (x));
double a := a*2;

add x y := x + y;
inc := add (1);
eleven := inc (10);

fourty := twice (double, 10);


main := div (999, 2);
```

E-2 Submission and grading

E-2.1 Rules and instructions

- Both students in a group are expected to contribute to all parts of the project, and individual students can be asked to explain all code in an additional oral examination while the projects are graded.
- Without *all* the mandatory features you cannot receive a grade above 4.
- Submission should *strictly* follow the submission instructions (Section E-2.4). Up-to one full point could be subtracted when they are not strictly followed. Any submission that severely deviates from the instructions is considered as “no submission”, and consequently not graded. Make sure to use `MakeZip.hs` and `CheckZip.hs` (both from `FP-Project.zip`) before submission.

- All features are in the correct file, and comments clearly indicate where a feature is implemented using the feature identifier (**use:** `-- FPxx.yy`). If we do not easily find a feature or cannot easily test it, no points are rewarded for it.
- All features (FPXX.yy) have the type as indicated in this assignment, have useful comments, and at least one function (not commented out!) to demonstrate how the feature is used (e.g., for FP1.5 you can demonstrate `pure` with `pureEx = runParser (pure 42) (Stream "123")` and `<*>` with `appEx = runParser ((,) <$> char 'a' <*> char 'b') (Stream "ab")`). Otherwise the assessor may not be able to (fully) assess your implementation.
- All submissions are checked for code similarity (both manually and using software) and for fraud. Suspicion of fraud or enabling it (e.g., sharing solutions in a open repository) is always reported to the Board of Examiners. For fraud we use the definition below.
- For some parts the TA's will discuss the project during the supervised sessions (see Section E-2.2). Please share your progress with them and ask for some feedback such that we can address misunderstandings or problems early on. Please take the initiative to do so latest 04-06-2021.

Definition of fraud, from the Faculty of Applied Science (TUD):

> Intentional acts or omissions on the part of a student, which render correct or fair evaluations of his/her knowledge, insight of skills totally or partially impossible.

E-2.2 Assessment

The grading of your work is based on the following criteria:

- Readability and style of the Haskell code,
- Use of the appropriate functional programming concepts and constructs,
- Use of the appropriate abstractions, e.g., higher order functions, lambda abstraction, type classes and (Applicative) Functors,
- An indicator `-- FPxx.yy`, comments, tests, type annotations and examples should be available for all functions and features,
- Features (indicated with FP1.1 to FP5.6); the *maximum* points are indicated in the tables within the sections E-3 to E-7.

The grade is calculated as follows:

$$\text{grade} = \max(\min(100, x), 10)/10,$$

where x is the number of points you received (again, see the tables below). In order to give you some flexibility in what topics to focus on, you can earn more than 100 points. Use this flexibility wisely because of the assessment criteria above. Note, that it may be better to omit features if otherwise the readability of your code as a whole would suffer.

> **Important:** before you submit your work, please ask feedback to the TAs during the supervised project sessions about the following (latest 04-06-2021 to have time to process feedback):
>
> - The definition of the EDSL (FP3.1),
> - The set of mandatory features,
> - When you choose to implement error handling: show the data types you have in mind for FP5.1.
>
> Note, that this check means that the TA is aware of your progress. It is not an indication that your code is correct.

E-2.3 Suggested and mandatory features

Some of the features are mandatory since these are essential for (almost) all other features and the structure of the project (see Table E.1). Some of the optional features are (strongly!) suggested to implement, since

	Features
Mandatory	FP1.1, FP1.2, FP1.3, FP1.5, FP1.6, FP2.2, FP3.1, FP3.2, FP4.1
Strongly suggested	FP1.4, FP2.1, FP2.3, FP2.4, FP3.3, FP4.2

Table E.1: Mandatory and suggested features

they make later exercises significantly easier (see Table E.1). Section E-7 contains some extensions that are best postponed until all the suggested features are implemented. The tables with features below indicate next to the maximally awarded points also if a feature is mandatory (M) or strongly suggested (S).

E-2.4 Available code and submission

On Canvas you can find the file `FP-Project.zip`, which contains files that you need to change. You may use all the code in these files, and also GHC, twentefp-eventloop-trees, and the entire Prelude (the standard libraries) except for the parser combinator functions and Monads. Note, using ParSec, other predefined parser combinators and Monad related syntax (including do-notation and `>>=`) is therefore not allowed.

Since the files are processed automatically, do not rename the files, take care of file name capitalization, etc. At submission, use .zip (**important**: PKZip — not another compression format) with *exactly* the following directory structure and files:

- `sxxxxxxxx_syyyyyyyy/PComb.hs`
- `sxxxxxxxx_syyyyyyyy/BasicParsers.hs`
- `sxxxxxxxx_syyyyyyyy/MicroFP.hs`

Here `sxxxxxxxx` and `syyyyyyyy` should be replaced by the student numbers of the two students in the group (use `sxxxxxxxx` instead of `sxxxxxxxx_syyyyyyyy` when you work alone). For your convenience, you may consider using `MakeZip.hs` from `FP-Project.zip` to create the .zip-file. Always use `CheckZip.hs` from `FP-Project.zip` to check some of the basic submission requirements. Check in time if this tool works for you and do not submit if it does not pass the test. If your submissions does not follow the guidelines it is either considered as “no submission”, or points are deducted (see above).

E-3 Parser combinator library

In `PComb.hs` you can find some example code for parser combinators. Change/add/extend the parser combinator library (`PComb.hs`) to support (some of) the features below:

	Points	Description
FP1.1	2 (M)	The parser can receive a "stream" (see `Stream`) of **`Char`**s and result in some type `a`. This implies that a parser is of type `Parser a`, where `a` is the type of the parse result.
FP1.2	3 (M)	The parser has an appropriate **`Functor`** instance.
FP1.3	1 (M)	The function `char :: Char -> Parser Char` parses a single (given) **`Char`**.
FP1.4	2 (S)	The function `failure :: Parser a` is a parser that consumes no input and fails (produces no valid parsing result).
FP1.5	2 (M)	The parser has an `Applicative` instance for the sequence combinator.
FP1.6	2 (M)	The parser has an `Alternative` instance that tries as few alternatives as possible (i.e., until the first parser succeeds).
Total	**12**	

Hints/suggestions:

- FP1.6: `many` and `some`: look at how they are recursively defined (for this, look them up on Hoogle). The definitions rely on lazy evaluation, while pattern matching constructors enforces evaluation. You can work around this by using `runParser`.

E-4 Basic Parsers

Next to the core parser combinators, several basic parsers/combinators are needed to make our parsing library useful. To accomplish this, define the following data types and functions in `BasicParsers.hs`. Make sure to use already defined parsers and combinators, instead of operating on the input stream directly (this results in deducted, or even zero points).

	Points	Description
FP2.1	5 (S)	Define the parsers `letter :: Parser Char` that parses any (alphabetical) letter, and `dig :: Parser Char` that parses any digit.
FP2.2	2 (M)	The following parser combinators: [1pt] `between :: Parser a -> Parser b -> Parser c -> Parser b` runs the three parsers in sequence, and returns the result of the second parser. Similar to `between` in ParSec. [1pt] `whitespace :: Parser a -> Parser a` receives a parser `p` and uses it to parse the input stream, while skipping all surrounding whitespaces (space, tab and newline). For example, `whitespace (char 'a')` can be used to parse the input `"␣a␣a␣"` and then results in `"a␣a"`.
FP2.3	3 (S)	The following parser combinators: [1pt] `sep1 :: Parser a -> Parser b -> Parser [a]`. The parser `sep1 p s` parses one or more occurrences of `p`, separated by `s`. This can, for example, be used to parse a comma separated list. [1pt] `sep :: Parser a -> Parser b -> Parser [a]`. The parser `sep p s` works as `sep1 p s`, but parses zero or more occurrences of `p`. [1pt] `option :: a -> Parser a -> Parser a`. `option x p` tries to apply parser `p`; upon failure it results in `x`. Similar to `option` in ParSec.
FP2.4	10 (S)	The following parsers and combinators: [3pt] `string :: String -> Parser String` parses an given `String`, similar to the function `char`. [2pt] `identifier :: Parser String` parses an identifier surrounded by whitespace. [2pt] `integer :: Parser Integer` parses an integer surrounded by whitespace. [1pt] `symbol :: String -> Parser String` parses a given `String` surrounded by whitespaces. [1pt] `parens :: Parser a -> Parser a` parses something using the provided parser between parentheses, i.e., `(...)`. [1pt] `braces :: Parser a -> Parser a` parses something using the provided parser between braces.
Total	**20**	

E-5 Embedded Domain Specific Language

Define an EDSL and interpreter that correspond to the μFP grammar by defining at least the following data types and functions in `MicroFP.hs`:

	Points	Description
FP3.1	5 (M)	A set of type/data constructors that represent an EDSL (deep embedding) for μFP, i.e., it describes the same functionality. Note, that this EDSL can be made very compact compared to the grammar. Define the type constructor `Prog` to indicate the type of a μFP program (top level), such that we can easily recognise your types. Be creative in your design!
FP3.2	2 (M)	Define the following functions in your μFP EDSL, all of type `Prog` (i.e., a program with one function). These must correspond to the definitions in `functions.txt` • `fibonacci` receives a single argument "n" (note this is a μFP argument, not a Haskell argument! The same applies below), and expresses the calculation of the n-th fibonacci number. • `fib` receives a single argument "n", and expresses the calculation of the n-th fibonacci number. • **`sum`** receives one argument "a", and calculates the sum from 1 to a • **`div`** receives two arguments "x" and "y", and divides "x" by "y" • `twice` receives two arguments"f" (a function) and "x" (a value), and calculates $f(f(x))$ • A data structure that describes `add`, `inc` and `eleven` from `functions.txt`
FP3.3	3 (S)	`pretty :: ? ->` **`String`** is a pretty printer that generates a textual representation that corresponds to the grammar of μFP. Here `?` corresponds to your EDSL.
FP3.4	15	`eval : Prog ->` **`String`** `-> [`**`Integer`**`] ->` **`Integer`**, which is a evaluator for your μFP EDSL *without* support for partial application, lazy evaluation, pattern matching and higher order functions. The function with the given name (of type **`String`**) in a program (of type `Prog`) is evaluated with the arguments of type `[`**`Integer`**`]`, resulting in an **`Integer`**. Since pattern matching is not (yet) supported, you may assume constants are not used at the left-hand side (e.g., `fibonacci` does not work with your evaluator yet). We will test your evaluator with some of the definitions from FP3.1 (e.g., `eval fibonacci [10]`), make sure they work with `eval`.
Total	**25**	

Hints/suggestions:

- FP3.3: Define a type class for pretty printing.
- FP3.4: Define a function `bind` that receives an expression, a list of (variable) names and a list of values/expressions. The function replaces each occurrence of a variable by the corresponding value/-expression.
- FP3.4: Define a function `reduce` that receives an expression and reduces it to a simpler expression, and uses `bind` where needed.

E-6 Parser

Define a parser that translates a textual representation of μFP to your μFP EDSL. For this, define the following functions in `MicroFP.hs`, where `?` corresponds to one or more types from your μFP EDSL:

	Points	Description
FP4.1	10 (M)	Define the parsers `factor :: Parser ?`, `expr :: Parser ?`, `term :: Parser ?` and the remaining parsers needed to parse μFP (here `?` is a type from your EDSL). Make sure `functions.txt` is parsed properly and leads to similar definitions as in FP3.2.
FP4.2	5 (S)	`compile :: String -> ?` tokenizes and compiles a textual representation of μFP to your EDSL.
FP4.3	5	`runFile :: FilePath -> [Integer] -> IO Integer` results in an `IO` action that reads the specified file, compiles it, and finally uses `eval` (from FP3.4) to evaluate it with the list of integers as input to the μFP function. When the file contains multiple functions, use the *last* function in the file.
Total	**20**	

E-7 Further features

	Points	Description
FP5.1	15	Implement error handling as follows: • A parsing failure results in an error of the form: "Parse error at 4:3, expected 'a"', where 4 is the line number and 3 the column where the error occurred, • expected alternatives can be reported (e.g., when `<|>` is used), for example: "Parse error at 4:3, expected 'a' or 'b"', • a parser combinator `(<?>) :: Parser a -> String -> Parser a` that creates error messages; as is also available in ParSec, • appropriate changes to all the tokenizer code to support error handling, • when parsing fails, `compiler` uses `error` to report the error message. Note that error handling has a major impact on every aspect of the project.
FP5.2	7	Implement pattern matching for `eval`. The μFP functions `sum` and `fibonacci` must work correctly.
FP5.3	3	Implement the function `patmatch`, which rewrites function definitions with pattern matching (in your EDSL) to a single definition with `if`-expressions. The μFP functions `sum` and `fibonacci` must work correctly.
FP5.4	5	Implement partial application for `eval`. The μFP function `inc` must work correctly.
FP5.5	5	Implement higher order functions for `eval`. Now the μFP function `twice` works correctly **(FP5.4 must be implemented first)**.
FP5.6	5	Test your compiler framework using QuickCheck as follows: • Make your EDSL an instance of `Arbitrary` • Generated function names are strings of length 3 containing 'f', 'g', 'h' and/or 'k'. Generated identifiers are strings of length 3 containing 'a', 'b', 'c', 'd' and/or 'e'. • Define a QuickCheck property that checks if a randomly generated EDSL expression, after pretty printing and compilation, remained the same. Make sure the data structures have a variable, but moderate size. • Define a QuickCheck property that checks if a randomly generated EDSL application/program, after pretty printing and compilation, remained the same. Make sure the data structures have a variable, but moderate size.
Total	**40**	