# MOD08: Functional Programming

## Functional Programming Project

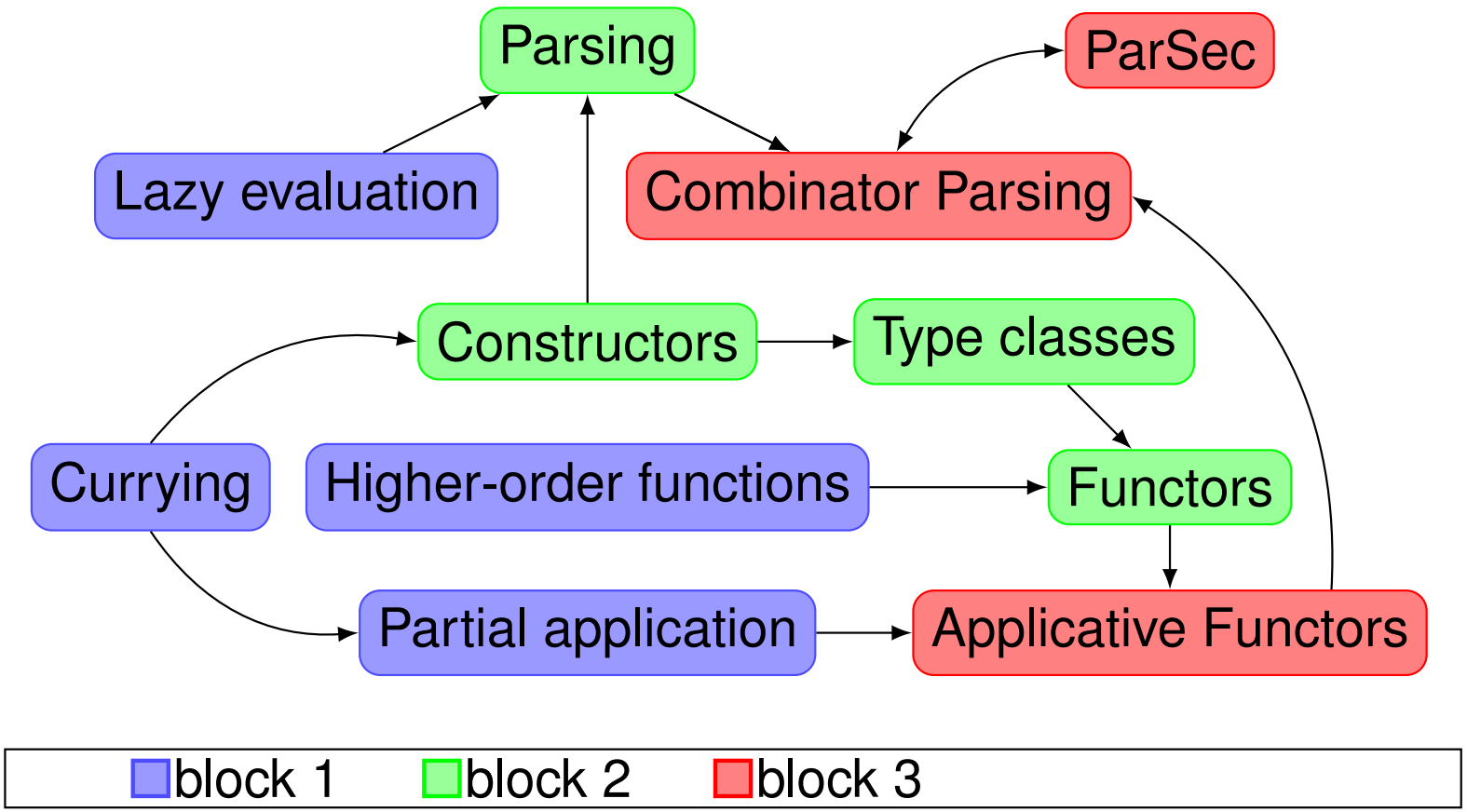
Marco Gerards

May 2019

# Goal of this lecture

- Introduce the main project goals
- Introduce the subgoals and give some hints
- Explanation of a QuickCheck feature you may use
- Organizational aspects

# Connection of some of the topics between blocks



# $\mu$FP

⟨*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*⟩ ::='<' | '==' | '>'

# $\mu$FP examples (1/2)

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

# $\mu$FP examples (1/2)

```
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)
         };
```

# $\mu$FP examples (1/2)

```
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;
```

# $\mu$FP examples (2/2)

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

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

# $\mu$FP partial application and higher-order functions

```
twice f x := f (f (x));
double a := a*2;
fourty := twice (double, 10);
```

# $\mu$FP partial application and higher-order functions

```
twice f x := f (f (x));
double a := a*2;
fourty := twice (double, 10);


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

# FP Project overview

- Topics:
  - Define your own parser combinator library **(20 points)**

# FP Project overview

- Topics:
  - Define your own parser combinator library **(20 points)**
  - Create several basic parsers **(20 points)**

# FP Project overview

- Topics:
  - Define your own parser combinator library **(20 points)**
  - Create several basic parsers **(20 points)**
  - Create an EDSL with the $\mu$FP functionality **(25 points)**

# FP Project overview

- Topics:
  - Define your own parser combinator library **(20 points)**
  - Create several basic parsers **(20 points)**
  - Create an EDSL with the $\mu$FP functionality **(25 points)**
  - Create a $\mu$FP parser with as target your EDSL **(20 points)**

# FP Project overview

- Topics:
  - Define your own parser combinator library **(20 points)**
  - Create several basic parsers **(20 points)**
  - Create an EDSL with the $\mu$FP functionality **(25 points)**
  - Create a $\mu$FP parser with as target your EDSL **(20 points)**
  - Further (optional) features **(40 points)**
    - Example: test your parser using QuickCheck

# FP Project overview

- Topics:
  - Define your own parser combinator library **(20 points)**
  - Create several basic parsers **(20 points)**
  - Create an EDSL with the $\mu$FP functionality **(25 points)**
  - Create a $\mu$FP parser with as target your EDSL **(20 points)**
  - Further (optional) features **(40 points)**
    - Example: test your parser using QuickCheck
- Grade:

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

- In total **125** points; not all features are mandatory

# FP Project overview

- Topics:
  - Define your own parser combinator library **(20 points)**
  - Create several basic parsers **(20 points)**
  - Create an EDSL with the $\mu$FP functionality **(25 points)**
  - Create a $\mu$FP parser with as target your EDSL **(20 points)**
  - Further (optional) features **(40 points)**
    - Example: test your parser using QuickCheck
- Grade:

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

- In total **125** points; not all features are mandatory
- Read the assignment for details about grading and points

# Material

- Canvas: zip-file containing
  - `PComb.hs` for the parser combinator
  - `BasicParsers.hs` for the tokenizer
  - `MicroFP.hs` for the EDSL, evaluator, parser and QuickCheck

# Material

- Canvas: zip-file containing
  - `PComb.hs` for the parser combinator
  - `BasicParsers.hs` for the tokenizer
  - `MicroFP.hs` for the EDSL, evaluator, parser and QuickCheck

- You may use
  - GHC
  - The Prelude, except for parser combinators
  - QuickCheck
  - twentefp-eventloop-trees

# Material

- Canvas: zip-file containing
  - `PComb.hs` for the parser combinator
  - `BasicParsers.hs` for the tokenizer
  - `MicroFP.hs` for the EDSL, evaluator, parser and QuickCheck
- You may use
  - GHC
  - The Prelude, except for parser combinators
  - QuickCheck
  - twentefp-eventloop-trees
- Do not use Parsec or other libraries / parser combinators

# Parser definition

- Given parser:

```
data Stream = Stream [Char]
              deriving Show

data Parser r = P {
  runParser :: Stream -> [(r, Stream)]
}
```

# Parser definition

- Given parser:

```
data Stream = Stream [Char]
              deriving Show

data Parser r = P {
  runParser :: Stream -> [(r, Stream)]
}
```

- Lacks:
  - Parsers and parser combinators
    - <$>, <*>, <|>
    - `some` and `many` (define an Alternative instance!)
    - `char`, `integer`, `symbol`, etc.
  - Error handling

# Alternatives

- Lecture on parser combinators: `<|>` gives *all* possible ways of parsing
- Parsec: `p <|> q` tries parser `p` first, when nothing is consumed it tries `q`

# Alternatives

- Lecture on parser combinators: `<|>` gives *all* possible ways of parsing
- Parsec: `p <|> q` tries parser `p` first, when nothing is consumed it tries `q`
- In the project:
  - `p <|> q` tries parser `p` first, on failure it tries `q`
    - Do not use your practicum code: it behaves differently
  - `p <> q` gives *all* possible ways of parsing (uses both `p` *and* `q`)
    - Use `Monoids`
    - *Almost* how Parsec works
    - `mconcat` can now be used to define other parsers/combinators

# Embedded Domain Specific Language

- Describe $\mu$FP using an EDSL (deep embedding)

# Embedded Domain Specific Language

- Describe $\mu$FP using an EDSL (deep embedding)
- Aim for a compact description
  - You have some design freedom
  - Keep evaluation and parsing in mind

# Embedded Domain Specific Language

- Describe $\mu$FP using an EDSL (deep embedding)
- Aim for a compact description
  - You have some design freedom
  - Keep evaluation and parsing in mind
- Define an evaluator
  - Simple: no pattern matching, laziness, partial application and higher order functions
  - Optional extension: pattern matching
  - Optional extension: partial application and higher order functions

# Embedded Domain Specific Language

- Describe $\mu$FP using an EDSL (deep embedding)
- Aim for a compact description
  - You have some design freedom
  - Keep evaluation and parsing in mind
- Define an evaluator
  - Simple: no pattern matching, laziness, partial application and higher order functions
  - Optional extension: pattern matching
  - Optional extension: partial application and higher order functions
- Define a function `pretty` for pretty printing
  - Inverse of compilation

# QuickCheck

- How to test if the compiler is correct?

# QuickCheck

- How to test if the compiler is correct?
- Approach:
  1. Generate a random program `p` (EDSL)
  2. Pretty print it
  3. Compile it to `p'`
  4. check if `p == p'`

# QuickCheck

- How to test if the compiler is correct?
- Approach:
  1. Generate a random program `p` (EDSL)
  2. Pretty print it
  3. Compile it to `p'`
  4. check if `p == p'`
- How to generate random programs?

# QuickCheck

- How to test if the compiler is correct?
- Approach:
  1. Generate a random program `p` (EDSL)
  2. Pretty print it
  3. Compile it to `p'`
  4. check if `p == p'`
- How to generate random programs?
  - Provide an `Arbitrary` instance

# Organisation

- Work in pairs; same pairs as in the practical sessions
- Indicate in your code where you implement a certain feature
  - Features are numbered from FP1.1 to FP5.6
  - See the project description
- Your work is processed using scripts:
  - Submit a PKZip (.zip-file); no 7Zip/RAR/tar/etc.
  - Create a directory `sxxxxxxxx_syyyyyyyy`
    - Replace `sxxxxxxxx` and `sxxxxxxxx` by your student numbers!
  - In this directory add: `PComb.hs`, `BasicParsers.hs` and `MicroFP.hs`
  - Name of zip: `sxxxxxxxx_syyyyyyyy.zip`
  - When you work alone: `sxxxxxxxx.zip` (directory: `sxxxxxxxx`)
  - `MakeZip.hs` may be used; no support, you remain responsible
  - Submit only once per group (by one student)
- Points are deducted for not following instructions

# MakeZip.hs

- MakeZip.hs: checks .hs files, and build a zip-file
- In `PComb.hs`, `BasicParsers.hs` *and* `MicroFP.hs`:

```
-- Student 1: Your Name (sxxxxxxx)
-- Student 2: Other Name (syyyyyyy)
```

# MakeZip.hs

- MakeZip.hs: checks .hs files, and build a zip-file
- In `PComb.hs`, `BasicParsers.hs` *and* `MicroFP.hs`:

```
-- Student 1: Your Name (sxxxxxxx)
-- Student 2: Other Name (syyyyyyy)
```

- If Haskell Curry (stud. nr. s1234567) works with Ada Lovelace (stud. nr. s9876543):

```
-- Student 1: Haskell Curry (s1234567)
-- Student 2: Ada Lovelace (s9876543)
```

# MakeZip.hs

- MakeZip.hs: checks .hs files, and build a zip-file
- In `PComb.hs`, `BasicParsers.hs` *and* `MicroFP.hs`:

```
-- Student 1: Your Name (sxxxxxxx)
-- Student 2: Other Name (syyyyyyy)
```

- If Haskell Curry (stud. nr. s1234567) works with Ada Lovelace (stud. nr. s9876543):

```
-- Student 1: Haskell Curry (s1234567)
-- Student 2: Ada Lovelace (s9876543)
```

- If Haskell Curry works alone:

```
-- Student 1: Haskell Curry (s1234567)
-- Student 2: Other Name (syyyyyyy)
```

# MakeZip.hs

- MakeZip.hs: checks .hs files, and build a zip-file
- In `PComb.hs`, `BasicParsers.hs` *and* `MicroFP.hs`:

```
-- Student 1: Your Name (sxxxxxxx)
-- Student 2: Other Name (syyyyyyy)
```

- If Haskell Curry (stud. nr. s1234567) works with Ada Lovelace (stud. nr. s9876543):

```
-- Student 1: Haskell Curry (s1234567)
-- Student 2: Ada Lovelace (s9876543)
```

- If Haskell Curry works alone:

```
-- Student 1: Haskell Curry (s1234567)
-- Student 2: Other Name (syyyyyyy)
```

- Load `MakeZip.hs` in GHCi, evaluate `main` and follow the instructions to get a zip-file

# MakeZip.hs

- MakeZip.hs: checks .hs files, and build a zip-file
- In `PComb.hs`, `BasicParsers.hs` *and* `MicroFP.hs`:

```
-- Student 1: Your Name (sxxxxxxx)
-- Student 2: Other Name (syyyyyyy)
```

- If Haskell Curry (stud. nr. s1234567) works with Ada Lovelace (stud. nr. s9876543):

```
-- Student 1: Haskell Curry (s1234567)
-- Student 2: Ada Lovelace (s9876543)
```

- If Haskell Curry works alone:

```
-- Student 1: Haskell Curry (s1234567)
-- Student 2: Other Name (syyyyyyy)
```

- Load `MakeZip.hs` in GHCi, evaluate `main` and follow the instructions to get a zip-file
- No support is given on `MakeZip.hs`
  - if it does not work, make the zip by hand!
  - check the zip-file, you are responsible for the content

# Plagiarism

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.

All submissions are checked for plagiarism and code similarity

# Grading and further requirements

- Some features are mandatory

# Grading and further requirements

- Some features are mandatory
- Some features are (strongly) suggested

# Grading and further requirements

- Some features are mandatory
- Some features are (strongly) suggested
- This be signed-off (deadline: 05-06-2019):
  1. Definition of the EDSL (FP3.1)
  2. The set of mandatory features
  3. When error handling is implemented: data types for FP5.1

# Grading and further requirements

- Some features are mandatory
- Some features are (strongly) suggested
- This be signed-off (deadline: 05-06-2019):
  1. Definition of the EDSL (FP3.1)
  2. The set of mandatory features
  3. When error handling is implemented: data types for FP5.1
- Sign-off means: feature is implemented, perhaps incorrectly

# Grading and further requirements

- Some features are mandatory
- Some features are (strongly) suggested
- This be signed-off (deadline: 05-06-2019):
  1. Definition of the EDSL (FP3.1)
  2. The set of mandatory features
  3. When error handling is implemented: data types for FP5.1
- Sign-off means: feature is implemented, perhaps incorrectly
- Assessment criteria
  - Readability
  - Appropriate FP concepts and constructs
  - Appropriate abstractions (HO functions, type classes, Applicative Functors, Monoids, etc.)
  - Comments and small tests
  - Features

# Grading and further requirements

- Some features are mandatory
- Some features are (strongly) suggested
- This be signed-off (deadline: 05-06-2019):
  1. Definition of the EDSL (FP3.1)
  2. The set of mandatory features
  3. When error handling is implemented: data types for FP5.1
- Sign-off means: feature is implemented, perhaps incorrectly
- Assessment criteria
  - Readability
  - Appropriate FP concepts and constructs
  - Appropriate abstractions (HO functions, type classes, Applicative Functors, Monoids, etc.)
  - Comments and small tests
  - Features
- Preferred: quality over quantity

# Submission

- Submission deadline: June 7th, 23:59
- Late submissions: points are deducted (module guide, Ch. 1)
- Submissions via Canvas only