Programming in Haskell

Jan Šnajder

University of Zagreb
Faculty of Electrical Engineering and Computing

Academic Year 2015/2016

Creative Commons Attribution–NonCommercial-NoDerivs 3.0

v1.0

Outline

Outline

PUH – Teaching Staff

hazel dormouse *(muscardinus avellanarius)*

Lecturer:

- Jan Šnajder
  jan.snajder@fer.hr

PUH – TA Squad

- Antonio Paunović
  antonio.paunovic@fer.hr
- Ivan Paljak
  ivan.paljak@fer.hr
- Luka Horvat
  lukahorvat9@gmail.com
- Luka Skukan
  luka.skukan@fer.hr
- Teon Banek
  theongugl@gmail.com

What is PUH about?

An introductory course to programming in advanced, purely functional programming language Haskell.

We assume that...

- you don't have a clue about Haskell
- you've passed a course on basic programming (e.g., PIPI) and algorithms+data structures (e.g., ASP)
- you're HIGHLY motivated to learn Haskell
- you'll program at least 8 hours a week in Haskell
- you'll devote about 1 hour a week for peer-reviewing

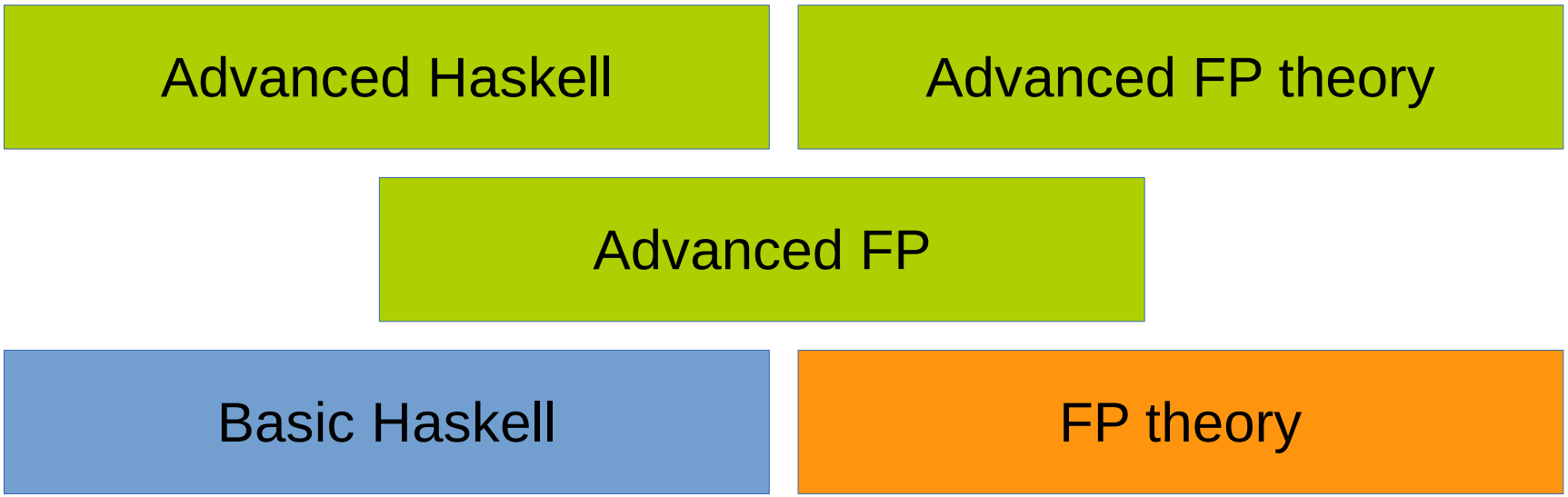
The Cunning Plan

1. Quick intro to FP & Haskell
2. Getting started
3. Basic language constructs, tuples and lists
4. Types and type classes
5. Syntax of functions
6. Recursive functions
7. Higher-order functions
8. Custom data types
9. Built-in datatypes, custom type classes
10. Input-output operations
11. Monads and monadic programming
12. Extras: Overview of most important packages

So-called learning outcomes

1. Define the basic concepts of functional programming in Haskell
2. Explain the syntax and semantics of a Haskell program
3. Use Haskell to solve simple practical problems
4. Apply functional idioms and functional design patterns
5. Use available programming libraries to solve complex problems
6. Compare alternative Haskell programs to determine which are better according to selected criteria
7. Design Haskell programs

"Haskell+FP stack"



- PUH covers: Basic Haskell programming
- What PUH is not about: FP theory (typed lambda calculus, functional data structures), advanced FP (functional algorithm design, functional problem solving), advanced Haskell programming (optimization, arrows, parallel programming, generics, extensions,...), advanced FP theory (category theory)

The bitter truth about programming

The only way to learn programming in Haskell is to. . . program :-)

Programming in PUH

1. In-class exercises
2. Homework assignments
3. Project assignment

Homework assignments

- Weekly assignments, about 10 in total, handed out on Fridays (or Saturday, or Sunday...)
- Contribute 70% to the final score
- Each assignment is scored on a scale from 1 to 10
- Submissions via Ferko before the deadline (Thursdays, 17:00)
- You can revise your submission as many times as you want before the deadline (do not "lock" your submission, why would you?)
- Late upload: possible only once, within 24 hours after the deadline, with no penalty
- Your HA will be checked automatically for correctness and additionally reviewed by a TA and two of your peers
- You'll receive a review pointing out the negative and positive aspects of your solution

Project assignment

- A small-sized but well-rounded application that does something useful
- Couple of modules, variable complexity in terms of supported features
- The assignment will be handed out after the midterm exams. You'll have 6 weeks to complete it
- Everyone gets the same assignment, unless you have a an absolutely amazing idea what you'd like to do and manage to convince us (do talk to us)

HA peer review

- Your task is to anonymously review two HAs
- Provide comments: what do you think of the solution and how it could be improved
- Useful for both parties: the other student gets a feedback, you get to see others' coding style

Course requirements

To pass this course, you'll need to...

1. Attend all lectures, except at most one
2. Submit all homework assignments within the prescribed deadlines
3. Score at least 50% on each homework assignment
4. Score at least 50% on the project assignment
5. NEW: Pass the midterm exam

Outline

Imperative vs. declarative programming

Programming languages: imperative and declarative

An imperative program is executed by (sequential) execution of commands, whereas a declarative program is executed by evaluating expressions

Declarative languages come in to flavors: logic languages (expressions are relations) and functional languages (expressions are functions)

Imperative PL: C, C++, C#, Java, ...

Declarative PL: SML, Ocaml, Haskell, F#, Erlang, Prolog, ...

Underlying these paradigms are different theoretical models of computation. The imperative paradigm is based on Turing machine, logic paradigm on formal logic, and functional paradigm on lambda calculus

Lambda calculus is expressively equivalent to a Turing machine and, according to Church-Turing thesis, describes the class of effectively computable functions (very fascinating, but let's leave it here)

Implicit vs. explicit state (1)

Imperative languages operate on an implicit state. Program execution amounts to changing this state by executing the individual commands

The change of state is typically accomplished by the assignment command

E.g., computing the factorial:

```
fact(x) {
  n = x;
  a = 1;
  while (n > 0) {
    a = a*n;
    n = n-1;
  }
  return a;
}
```

Implicit vs. explicit state (2)

Declarative languages have no implicit state

Program execution amounts to evaluation of expressions. There is no state involved, hence we call this state-less computation

There is no assignment operator (at least not as we know it)

Iteration is realized via recursion:

```
fact x = if x==0 then 1
         else x*fact(x-1)
```

State-full computation can be accomplished by introducing an explicit state, which we then need to drag around:

```
fact x = fact' x 1
  where fact' n a = if n>0 then fact' (n-1) (a*n)
                    else a
```

Referential transparency (1)

Declarative programming languages have the property of referential transparency: equally-valued expressions are substitutable in all contexts

```
... x+x ...
where x = fact 5
```

$\Rightarrow$ every occurrence of `x` in the program can be replaced with `fact 5`

This is not generally the case in imperative languages, because the value of variable `x` can be changed after being initialized

In imperative languages, we have to take account the order in which the commands are executed. Therefore, expressions in imperative languages are not referentially transparent

Side effect

Side effect: every change of the implicit state that violates the referential transparency of the program

```
foo(x) {
  y = 0;
  return 2*x;
}
```

```
...
y = 2
if (foo(y)==foo(y))
  then ... else ...
```

```
...
y = 5;
x = foo(2);
z = x + y
```

```
...
y = 5;
z = foo(2) + y;
...
```

Function `foo` has a side effect (y=0) and breaks the referential transparency in all contexts where the function is applied!

Referential transparency (2)

If we want to retain full referential transparency, we must not allow for any side affects!

What we get in return are programs that are:

1. formally concise
2. less error-prone
3. suitable for formal verification
4. suitable for parallelization


5x-2=13
then
x = 3
But professor, you said yesterday that x was equal to 2!

Functional programming (FP)

Programming style that is all about computing of functions

In FP, a function is a first-class citizen: it has the same *rights* as other data types. A function can be a return value or an argument of another function – higher-order function

```
twice f x = f (f x)
```

Other features of modern FP languages:

- lazy evaluation
- data abstraction
- pattern matching
- type inference

Functional programming languages

FP languages *support* and *encourage* FP style

Most popular FP languages:
Standard ML (SML), Clean, Miranda, Haskell, ...

Modern multiparadigmatic languages support FP:
Common Lisp, OCaml, Scala, Python, Ruby, F#, R, Mathematica, JavaScript, Clojure, ...

Some imperative languages support some FP concepts (C#, Java), while in some imperative languages FP concepts can be realized indirectly (C, C++)

Although many imperative languages support FP concepts, the purely functional subset of these languages is typically very weak!

Outline

Characteristics (1)

A purely functional language: side effects are banned completely, therefore programs are referentially transparent

Has lazy evaluation: avoids unnecessary computations and encourages modularization

Programs are concise and typically 2 to 10 times shorter than the same programs written in other programming languages

A powerful type system that is capable of detecting a large number of errors before run time

The type system supports both polymorphism and overloading

Less spectacular, but very useful: supports list comprehensions, a compact and expressive way of defining lists

Characteristics (2)

Emphasizes the use of recursion, and complements it with pattern matching and guards

Higher-order functions allow for high level of abstraction and the use of functional design patterns

Monadic programming allows for state-full computation (including IO) without giving up on referential transparency

A comprehensive Standard Library and a large number of additional modules (*Hackage*)

Haskell is developed and supported by a large community of scientists, professionals, and enthusiasts. Standardization is carried out by an international committee *(The Haskell Committee)*

Interpreters and compilers

- HUGS – interactive interpreter
  `http://www.haskell.org/hugs`
  Suitable for education and prototyping
- GHC (Glasgow Haskell Compiler) – interactive interpreter and compiler
  `http://www.haskell.org/ghc`
  Compiles into optimized C code, suitable for real-world applications

Outline

On-line materials

Haskell homepage

- http://www.haskell.org

Tutorials

- Haskell Wikibook
  http://en.wikibooks.org/wiki/Haskell
- Learn You A Haskell for Great Good!
  http://learnyouahaskell.com/
- Yet Another Haskell Tutorial
  http://darcs.haskell.org/yaht/yaht.pdf

Books

- Real World Haskell
  http://book.realworldhaskell.org/