CEng 3549 – Functional Programming

Histroy & Notions & A Taste of Haskell

Burak Ekici

September 22, 2022

About Me: Parcours/Carrier

Undergrad	IZTECH	CE	2004-2009, İzmir
Master's	Yaşar Üni	CE	2009-2012, İzmir
Traineeship	EC JRC	CE	2010-2011, Varese-Italy
PhD	U Joseph Fourier	CS & Math	2013-2015, Grenoble-France
PostDoc	U of Iowa	CS	2016-2017, IA-USA
PostDoc	U of Innsbruck	CS	2018-2019, Innsbruck-Austria
Assist. Prof. Dr.	Kültür Uni	CE	2019-2020, İstanbul
Assist. Prof. Dr.	TED Uni	CE	2021-2022, Ankara
Assist. Prof. Dr.	Muğla Sıtkı Koçman Uni	CE	2022-now, Muğla

Outline

Logistics

lecturer	Burak Ekici (burakekici@mu.edu.tr)

Logistics

lecturer	Burak Ekici (burakekici@mu.edu.tr)
teaching assistant	Erdem Türk (erdemturk@mu.edu.tr)

Logistics

lecturer	Burak Ekici (burakekici@mu.edu.tr)
teaching assistant	Erdem Türk (erdemturk@mu.edu.tr)
consultation	Thursday 13h30 – 16h30 at (no room assigned yet)

About the Course

- Prerequisites:
  - Strong motivation

About the Course

- Prerequisites:
  - Strong motivation
- Text Book:
  - Graham Hutton, Programming in Haskell, Cambridge University Press, 2007, ISBN 9780521692694.

About the Course

- Prerequisites:
  - Strong motivation
- Text Book:
  - Graham Hutton, Programming in Haskell, Cambridge University Press, 2007, ISBN 9780521692694.
- Additional References
  - Richard Bird, Introduction to Functional Programming using Haskell (2nd edition), Prentice Hall Europe, 1998, ISBN 0134843460.
  - Bryan O'Sullivan, Don Stewart, and John Goerzen, Real World Haskell, (freely available online) O'Reilly, 2008, ISBN 9780596514983.
  - Simon Thompson, Haskell: The Craft of Functional Programming, Addison-Wesley, 1996, ISBN 0201403579.
  - Chris Hankin, An Introduction to Lambda Calculi for Computer Scientists, King's College Publications, ISBN 0954300653.
  - Chris Okasaki, Purely Functional Data Structures, Cambridge University Press, 1999, ISBN 0521663504.
  - Fethi Rabhi and Guy Lapalme, Algorithms: A Functional Programming Approach, Addison-Wesley, 1999, ISBN 0201596040.

About the Course

- Prerequisites:
  - Strong motivation
- Text Book:
  - Graham Hutton, Programming in Haskell, Cambridge University Press, 2007, ISBN 9780521692694.
- Additional References
  - Richard Bird, Introduction to Functional Programming using Haskell (2nd edition), Prentice Hall Europe, 1998, ISBN 0134843460.
  - Bryan O'Sullivan, Don Stewart, and John Goerzen, Real World Haskell, (freely available online) O'Reilly, 2008, ISBN 9780596514983.
  - Simon Thompson, Haskell: The Craft of Functional Programming, Addison-Wesley, 1996, ISBN 0201403579.
  - Chris Hankin, An Introduction to Lambda Calculi for Computer Scientists, King's College Publications, ISBN 0954300653.
  - Chris Okasaki, Purely Functional Data Structures, Cambridge University Press, 1999, ISBN 0521663504.
  - Fethi Rabhi and Guy Lapalme, Algorithms: A Functional Programming Approach, Addison-Wesley, 1999, ISBN 0201596040.
- Tentative Grading:

~~Attendance~~	Homeworks	Midterm	Final
~~5%~~	35%	30%	35%

About the Course (cont'd: Goals – Roughly)

give an introduction to

- functional programming

About the Course (cont'd: Goals – Roughly)

give an introduction to

- functional programming
    - application examples based on Haskell (a pure and strict functional programming language)
    - theoretical background – λ-Calculus

About the Course (cont'd: Goals – Roughly)

give an introduction to

- functional programming
  - application examples based on Haskell (a pure and strict functional programming language)
  - theoretical background – λ-Calculus
- logical programming and type theory
  - techniques that allow for verification of functional programs
  - verification developments within the Coq proof assistant

Syllabus & Tentative Schedule

Week 0	History Notions & A Taste of Haskell

Syllabus & Tentative Schedule

Week 0	History Notions & A Taste of Haskell
Week 1	Type Classes & Lists & Patterns & Higher-Order Functions

Syllabus & Tentative Schedule

Week 0	History Notions & A Taste of Haskell
Week 1	Type Classes & Lists & Patterns & Higher-Order Functions
Week 2	Modules & Lists and Strings & Recursive Functions

Syllabus & Tentative Schedule

Week 0	History Notions & A Taste of Haskell
Week 1	Type Classes & Lists & Patterns & Higher-Order Functions
Week 2	Modules & Lists and Strings & Recursive Functions
Week 3	User-Defined Types & Trees & Input and Output

Syllabus & Tentative Schedule

Week 0	History Notions & A Taste of Haskell
Week 1	Type Classes & Lists & Patterns & Higher-Order Functions
Week 2	Modules & Lists and Strings & Recursive Functions
Week 3	User-Defined Types & Trees & Input and Output
Week 4	λ-Calculus

Syllabus & Tentative Schedule

Week 0	History Notions & A Taste of Haskell
Week 1	Type Classes & Lists & Patterns & Higher-Order Functions
Week 2	Modules & Lists and Strings & Recursive Functions
Week 3	User-Defined Types & Trees & Input and Output
Week 4	λ-Calculus
Week 5	Evaluation Strategies & Abstract Data Types & Sets as Binary Search Trees

Syllabus & Tentative Schedule

Week 0	History Notions & A Taste of Haskell
Week 1	Type Classes & Lists & Patterns & Higher-Order Functions
Week 2	Modules & Lists and Strings & Recursive Functions
Week 3	User-Defined Types & Trees & Input and Output
Week 4	λ-Calculus
Week 5	Evaluation Strategies & Abstract Data Types & Sets as Binary Search Trees
Week 6	Mathematical Induction & Induction over Lists & Structural Induction & The Coq Proof Assistant & Formal Verification of Functional Programs with Coq

Syllabus & Tentative Schedule

Week 0	History Notions & A Taste of Haskell
Week 1	Type Classes & Lists & Patterns & Higher-Order Functions
Week 2	Modules & Lists and Strings & Recursive Functions
Week 3	User-Defined Types & Trees & Input and Output
Week 4	λ-Calculus
Week 5	Evaluation Strategies & Abstract Data Types & Sets as Binary Search Trees
Week 6	Mathematical Induction & Induction over Lists & Structural Induction & The Coq Proof Assistant & Formal Verification of Functional Programs with Coq
Week 7	Midterm

Syllabus & Tentative Schedule

Week 0	History Notions & A Taste of Haskell
Week 1	Type Classes & Lists & Patterns & Higher-Order Functions
Week 2	Modules & Lists and Strings & Recursive Functions
Week 3	User-Defined Types & Trees & Input and Output
Week 4	λ-Calculus
Week 5	Evaluation Strategies & Abstract Data Types & Sets as Binary Search Trees
Week 6	Mathematical Induction & Induction over Lists & Structural Induction & The Coq Proof Assistant & Formal Verification of Functional Programs with Coq
Week 7	Midterm
Week 8	Efficiency & Tupling & Tail Recursion and Guarded Recursion & Property-Based Testing with LeanCheck

Syllabus & Tentative Schedule

Week 0	History Notions & A Taste of Haskell
Week 1	Type Classes & Lists & Patterns & Higher-Order Functions
Week 2	Modules & Lists and Strings & Recursive Functions
Week 3	User-Defined Types & Trees & Input and Output
Week 4	λ-Calculus
Week 5	Evaluation Strategies & Abstract Data Types & Sets as Binary Search Trees
Week 6	Mathematical Induction & Induction over Lists & Structural Induction & The Coq Proof Assistant & Formal Verification of Functional Programs with Coq
Week 7	Midterm
Week 8	Efficiency & Tupling & Tail Recursion and Guarded Recursion & Property-Based Testing with LeanCheck
Week 9	Parsing & Combinator Parsing & Parsing XML Data

Syllabus & Tentative Schedule

Week 0	History Notions & A Taste of Haskell
Week 1	Type Classes & Lists & Patterns & Higher-Order Functions
Week 2	Modules & Lists and Strings & Recursive Functions
Week 3	User-Defined Types & Trees & Input and Output
Week 4	λ-Calculus
Week 5	Evaluation Strategies & Abstract Data Types & Sets as Binary Search Trees
Week 6	Mathematical Induction & Induction over Lists & Structural Induction & The Coq Proof Assistant & Formal Verification of Functional Programs with Coq
Week 7	Midterm
Week 8	Efficiency & Tupling & Tail Recursion and Guarded Recursion & Property-Based Testing with LeanCheck
Week 9	Parsing & Combinator Parsing & Parsing XML Data
Week 10	Core FP & Type Checking & Unification and its Implementation & Type Inference

Syllabus & Tentative Schedule

Week 0	History Notions & A Taste of Haskell
Week 1	Type Classes & Lists & Patterns & Higher-Order Functions
Week 2	Modules & Lists and Strings & Recursive Functions
Week 3	User-Defined Types & Trees & Input and Output
Week 4	λ-Calculus
Week 5	Evaluation Strategies & Abstract Data Types & Sets as Binary Search Trees
Week 6	Mathematical Induction & Induction over Lists & Structural Induction & The Coq Proof Assistant & Formal Verification of Functional Programs with Coq
Week 7	Midterm
Week 8	Efficiency & Tupling & Tail Recursion and Guarded Recursion & Property-Based Testing with LeanCheck
Week 9	Parsing & Combinator Parsing & Parsing XML Data
Week 10	Core FP & Type Checking & Unification and its Implementation & Type Inference
Week 11	Laziness and Infinite Data Structures & Examples of (Infinite) Laziness

Syllabus & Tentative Schedule

Week 0	History Notions & A Taste of Haskell
Week 1	Type Classes & Lists & Patterns & Higher-Order Functions
Week 2	Modules & Lists and Strings & Recursive Functions
Week 3	User-Defined Types & Trees & Input and Output
Week 4	λ-Calculus
Week 5	Evaluation Strategies & Abstract Data Types & Sets as Binary Search Trees
Week 6	Mathematical Induction & Induction over Lists & Structural Induction & The Coq Proof Assistant & Formal Verification of Functional Programs with Coq
Week 7	Midterm
Week 8	Efficiency & Tupling & Tail Recursion and Guarded Recursion & Property-Based Testing with LeanCheck
Week 9	Parsing & Combinator Parsing & Parsing XML Data
Week 10	Core FP & Type Checking & Unification and its Implementation & Type Inference
Week 11	Laziness and Infinite Data Structures & Examples of (Infinite) Laziness
Week 12	Core FP Expressions & Implementing Type Inference

Syllabus & Tentative Schedule

Week 0	History Notions & A Taste of Haskell
Week 1	Type Classes & Lists & Patterns & Higher-Order Functions
Week 2	Modules & Lists and Strings & Recursive Functions
Week 3	User-Defined Types & Trees & Input and Output
Week 4	λ-Calculus
Week 5	Evaluation Strategies & Abstract Data Types & Sets as Binary Search Trees
Week 6	Mathematical Induction & Induction over Lists & Structural Induction & The Coq Proof Assistant & Formal Verification of Functional Programs with Coq
Week 7	Midterm
Week 8	Efficiency & Tupling & Tail Recursion and Guarded Recursion & Property-Based Testing with LeanCheck
Week 9	Parsing & Combinator Parsing & Parsing XML Data
Week 10	Core FP & Type Checking & Unification and its Implementation & Type Inference
Week 11	Laziness and Infinite Data Structures & Examples of (Infinite) Laziness
Week 12	Core FP Expressions & Implementing Type Inference
Week 13	An ‘Imperative‘ Evaluator & Monads & A Monadic Evaluator

Syllabus & Tentative Schedule

Week 0	History Notions & A Taste of Haskell
Week 1	Type Classes & Lists & Patterns & Higher-Order Functions
Week 2	Modules & Lists and Strings & Recursive Functions
Week 3	User-Defined Types & Trees & Input and Output
Week 4	λ-Calculus
Week 5	Evaluation Strategies & Abstract Data Types & Sets as Binary Search Trees
Week 6	Mathematical Induction & Induction over Lists & Structural Induction & The Coq Proof Assistant & Formal Verification of Functional Programs with Coq
Week 7	Midterm
Week 8	Efficiency & Tupling & Tail Recursion and Guarded Recursion & Property-Based Testing with LeanCheck
Week 9	Parsing & Combinator Parsing & Parsing XML Data
Week 10	Core FP & Type Checking & Unification and its Implementation & Type Inference
Week 11	Laziness and Infinite Data Structures & Examples of (Infinite) Laziness
Week 12	Core FP Expressions & Implementing Type Inference
Week 13	An ‘Imperative‘ Evaluator & Monads & A Monadic Evaluator
Week 14	Final

Outline

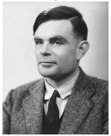
1936 **Alonzo Church:** λ-calculus

1918 2022

1936 **Alonzo Church:** λ-calculus

1918 2022

1937 **Alan Turing:** turing machines

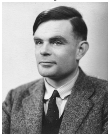


1936 **Alonzo Church:** λ-calculus

1924 **Moses Schönfinkel:** combinatory logic

1918 2022

1937 **Alan Turing:** turing machines



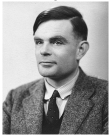
1936 **Alonzo Church:** λ-calculus

1924 **Moses Schönfinkel:** combinatory logic

1918 2022

1937 **Alan Turing:** turing machines

1930 **Haskell Curry:** combinatory logic



1924 **Moses Schönfinkel:** combinatory logic

1936 **Alonzo Church:** λ-calculus

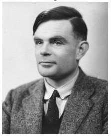
1941 **Z3:** 1st programmable, fully automatic computing machine

1918

2022

1930 **Haskell Curry:** combinatory logic

1937 **Alan Turing:** turing machines



1924 **Moses Schönfinkel:** combinatory logic

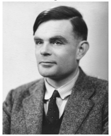
1936 **Alonzo Church:** λ-calculus

1941 **Z3:** 1st programmable, fully automatic computing machine

1918 2022

1930 **Haskell Curry:** combinatory logic

1937 **Alan Turing:** turing machines



1958 **John McCarthy:** LISP

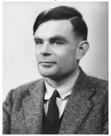
1918 — 2022

1924 **Moses Schönfinkel:** combinatory logic

1930 **Haskell Curry:** combinatory logic

1936 **Alonzo Church:** λ-calculus

1937 **Alan Turing:** turing machines



1941 **Z3:** 1st programmable, fully automatic computing machine

1958 **John McCarthy:** LISP

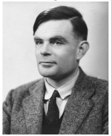
1966 **Peter Landin:** Iswim

1918 — 2022

1924 **Moses Schönfinkel:** combinatory logic

1930 **Haskell Curry:** combinatory logic

1936 **Alonzo Church:** λ-calculus



1937 **Alan Turing:** turing machines

1941 **Z3:** 1st programmable, fully automatic computing machine

1958 **John McCarthy:** LISP

1966 **Peter Landin:** Iswim

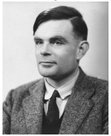
1977 **John Backus:** FP

1918 – 2022

1924 **Moses Schönfinkel:** combinatory logic

1930 **Haskell Curry:** combinatory logic

1936 **Alonzo Church:** λ-calculus



1937 **Alan Turing:** turing machines

1941 **Z3:** 1st programmable, fully automatic computing machine

1958 **John McCarthy:** LISP

1966 **Peter Landin:** Iswim

1977 **John Backus:** FP

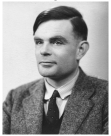
1984 **Robin Milner:** LCF, Standard ML

1918 — 2022

1924 **Moses Schönfinkel:** combinatory logic

1930 **Haskell Curry:** combinatory logic

1936 **Alonzo Church:** λ-calculus



1937 **Alan Turing:** turing machines

1941 **Z3:** 1st programmable, fully automatic computing machine

1958 **John McCarthy:** LISP

1966 **Peter Landin:** Iswim

1977 **John Backus:** FP

1984 **Robin Milner:** LCF, Standard ML

1985 **David Turner:** Miranda

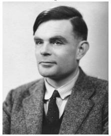
1918

1924 **Moses Schönfinkel:** combinatory logic

1930 **Haskell Curry:** combinatory logic

1936 **Alonzo Church:** λ-calculus

1937 **Alan Turing:** turing machines



1941 **Z3:** 1st programmable, fully automatic computing machine

1958 **John McCarthy:** LISP

1966 **Peter Landin:** Iswim

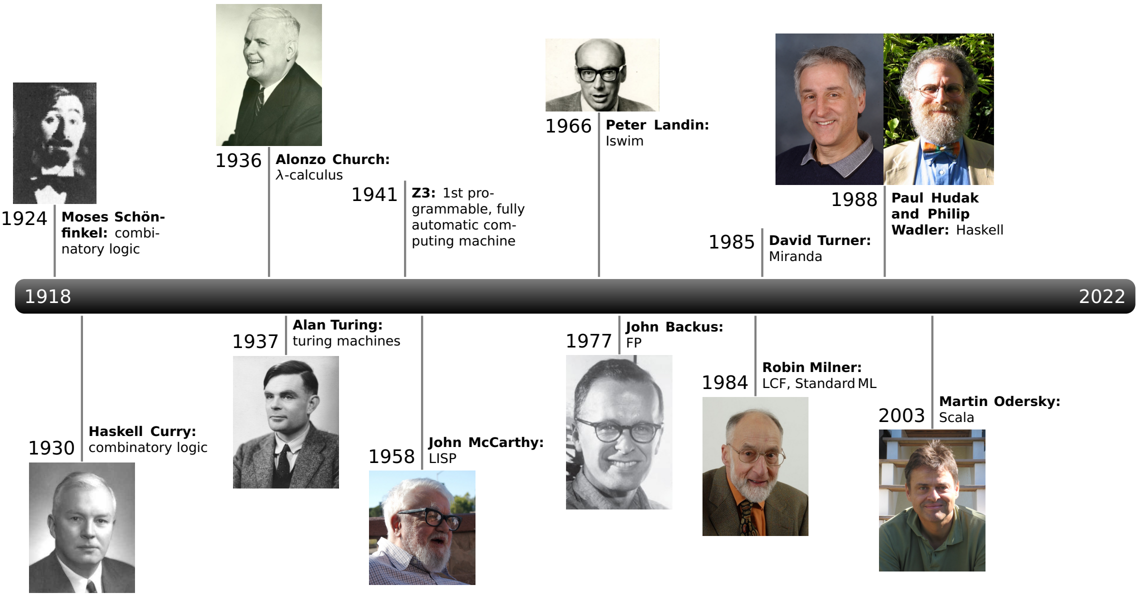
1977 **John Backus:** FP

1984 **Robin Milner:** LCF, Standard ML

1985 **David Turner:** Miranda

1988 **Paul Hudak and Philip Wadler:** Haskell

2022


1918
2022
1924 Moses Schönfinkel: combinatory logic
1930 Haskell Curry: combinatory logic
1936 Alonzo Church: λ-calculus
1937 Alan Turing: turing machines
1941 Z3: 1st programmable, fully automatic computing machine
1958 John McCarthy: LISP
1966 Peter Landin: Iswim
1977 John Backus: FP
1984 Robin Milner: LCF, Standard ML
1985 David Turner: Miranda
1988 Paul Hudak and Philip Wadler: Haskell
2003 Martin Odersky: Scala

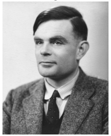
1918

1924 **Moses Schönfinkel:** combinatory logic

1930 **Haskell Curry:** combinatory logic

1936 **Alonzo Church:** λ-calculus

1937 **Alan Turing:** turing machines



1941 **Z3:** 1st programmable, fully automatic computing machine

1958 **John McCarthy:** LISP

1966 **Peter Landin:** Iswim

1977 **John Backus:** FP

1984 **Robin Milner:** LCF, Standard ML

1985 **David Turner:** Miranda


1988 **Paul Hudak and Philip Wadler:** Haskell

2003 **Martin Odersky:** Scala

2005 **Don Syme:** F#

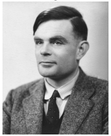
2022

1918 — 2022

1924 **Moses Schönfinkel:** combinatory logic

1930 **Haskell Curry:** combinatory logic

1936 **Alonzo Church:** λ-calculus



1937 **Alan Turing:** turing machines

1941 **Z3:** 1st programmable, fully automatic computing machine

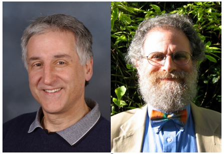
1958 **John McCarthy:** LISP

1966 **Peter Landin:** Iswim

1977 **John Backus:** FP

1984 **Robin Milner:** LCF, Standard ML

1985 **David Turner:** Miranda



1988 **Paul Hudak and Philip Wadler:** Haskell

2003 **Martin Odersky:** Scala

2005 **Don Syme:** F#

2010 **Haskell2010**

Outline

Definition ((program) state)

- variables point to storage locations in memory

Definition ((program) state)

- variables point to storage locations in memory
- state is content of variables in scope at given execution point

Definition ((program) state)

- variables point to storage locations in memory
- state is content of variables in scope at given execution point

Example (assignment)

after `x := 10`, location *x* has content 10 (state might have changed)

Definition ((program) state)

- variables point to storage locations in memory
- state is content of variables in scope at given execution point

Example (assignment)

after `x := 10`, location *x* has content 10 (state might have changed)

Side Effects

a function or expression has side effects if it modifies state

Definition ((program) state)

- variables point to storage locations in memory
- state is content of variables in scope at given execution point

Example (assignment)

after `x := 10`, location x has content 10 (state might have changed)

Side Effects

a function or expression has side effects if it modifies state

Example ($\sum_{i=0}^{n} i$)

```
count := 0
total := 0
while count < n
  count := count + 1
  total := total + count
```

Example ($\sum_{i=0}^{n} i$)

the Haskell way of summing up the numbers from `0` to `n` is

```
sum [0..n]
```

Example ($\sum_{i=0}^{n} i$)

the Haskell way of summing up the numbers from `0` to `n` is

```
sum [0..n]
```

- `[0..4]` generates list `[0,1,2,3,4]`
- `sum` is predefined function, summing up elements of a list

Example ($\sum_{i=0}^{n} i$)

the Haskell way of summing up the numbers from `0` to `n` is

```
sum [0..n]
```

- `[0..4]` generates list `[0,1,2,3,4]`
- `sum` is predefined function, summing up elements of a list

Example (defining functions)

- `[m..n]` computes range of numbers from `m` to `n`

```
range m n =
  if m > n then []
  else m : range (m + 1) n
```

Example ($\sum_{i=0}^{n} i$)

the Haskell way of summing up the numbers from `0` to `n` is

```
sum [0..n]
```

- `[0..4]` generates list `[0,1,2,3,4]`
- `sum` is predefined function, summing up elements of a list

Example (defining functions)

- `[m..n]` computes range of numbers from `m` to `n`

```
range m n =
  if m > n then []
  else m : range (m + 1) n
```

- `sum xs` computes sum of elements in `xs`

```
mySum [] = 0
mySum (x:xs) = x + mySum xs
```

Definition (pure functions)

a function is pure if it always returns same result on same input

Definition (pure functions)

a function is pure if it always returns same result on same input

Counterexample (random numbers)

the C function `rand` (producing random numbers) is not pure

```
rand() = 0
rand() = 10
rand() = 42
```

Definition (immutable data)

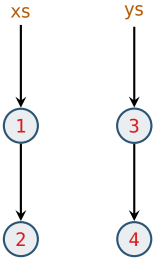
data that does not change after initial creation

Definition (immutable data)

data that does not change after initial creation

Example (immutable linked lists)

- consider two linked lists xs = [1,2] and ys = [3,4]
- after concatenation zs = xs ++ ys



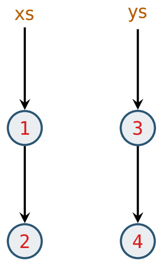
Definition (immutable data)

data that does not change after initial creation

Example (immutable linked lists)

- consider two linked lists xs = [1,2] and ys = [3,4]
- after concatenation zs = xs ++ ys

append elements of ys to xs

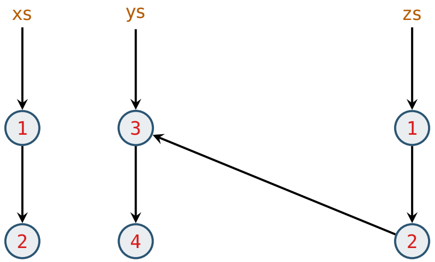


Definition (immutable data)

data that does not change after initial creation

Example (immutable linked lists)

- consider two linked lists xs = [1,2] and ys = [3,4]
- after concatenation zs = xs ++ ys

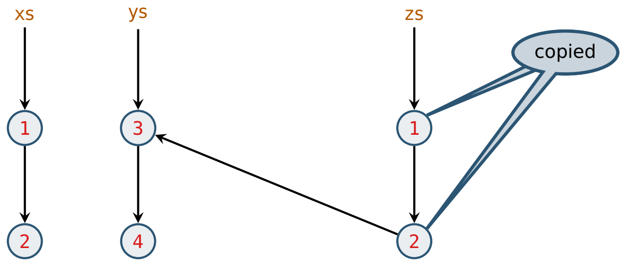


Definition (immutable data)

data that does not change after initial creation

Example (immutable linked lists)

- consider two linked lists xs = [1,2] and ys = [3,4]
- after concatenation zs = xs ++ ys



Recursion

a function (definition) is recursive if it refers to itself

Recursion

a function (definition) is recursive if it refers to itself

Example (factorial numbers)

```
factorial n =
  if n < 2 then 1
  else n * factorial (n - 1)
```

Evaluating Functions by Hand (aka Equational Reasoning)

- functions are defined by equations and pattern matching
- general idea: “replace equals by equals”

Evaluating Functions by Hand (aka Equational Reasoning)

- functions are defined by equations and pattern matching
- general idea: “replace equals by equals”

pattern: empty list

Example (mySum)

given the two equations

$$\mathtt{mySum}\ [\,] = 0 \tag{1}$$

$$\mathtt{mySum}\ (x{:}xs) = x + \mathtt{mySum}\ xs \tag{2}$$

Evaluating Functions by Hand (aka Equational Reasoning)

- functions are defined by equations and pattern matching
- general idea: “replace equals by equals”

Example (mySum)

pattern: list with “head” x and “tail” xs

given the two equations

```
mySum []     = 0                 (1)
mySum (x:xs) = x + mySum xs      (2)
```

Evaluating Functions by Hand (aka Equational Reasoning)

- functions are defined by equations and pattern matching
- general idea: "replace equals by equals"

Example (mySum)

given the two equations

$$\texttt{mySum}\ [\,] = 0 \tag{1}$$

$$\texttt{mySum}\ (x{:}xs) = x + \texttt{mySum}\ xs \tag{2}$$

we evaluate `mySum [1,2,3]` like

`mySum [1,2,3]`

Evaluating Functions by Hand (aka Equational Reasoning)

- functions are defined by equations and pattern matching
- general idea: “replace equals by equals”

Example (mySum)

given the two equations

$$\texttt{mySum}\ [\,] = 0 \tag{1}$$

$$\texttt{mySum}\ (x{:}xs) = x + \texttt{mySum}\ xs \tag{2}$$

we evaluate mySum [1,2,3] like

$$\texttt{mySum}\ [1,2,3] \quad = 1 + \texttt{mySum}\ [2,3] \qquad \text{using (2)}$$

Evaluating Functions by Hand (aka Equational Reasoning)

- functions are defined by equations and pattern matching
- general idea: "replace equals by equals"

Example (mySum)

given the two equations

$$\texttt{mySum}\ [\,] = \texttt{0} \tag{1}$$
$$\texttt{mySum}\ (\texttt{x}:\texttt{xs}) = \texttt{x} + \texttt{mySum}\ \texttt{xs} \tag{2}$$

we evaluate `mySum [1,2,3]` like

$$\begin{aligned} \texttt{mySum}\ [1,2,3] &= 1 + \texttt{mySum}\ [2,3] && \text{using (2)} \\ &= 1 + (2 + \texttt{mySum}\ [3]) && \text{using (2)} \end{aligned}$$

Evaluating Functions by Hand (aka Equational Reasoning)

- functions are defined by equations and pattern matching
- general idea: “replace equals by equals”

Example (mySum)

given the two equations

$$\texttt{mySum}\ [\,] = 0 \tag{1}$$

$$\texttt{mySum}\ (x{:}xs) = x + \texttt{mySum}\ xs \tag{2}$$

we evaluate `mySum [1,2,3]` like

$$\begin{aligned}
\texttt{mySum}\ [1,2,3] &= 1 + \texttt{mySum}\ [2,3] && \text{using (2)}\\
&= 1 + (2 + \texttt{mySum}\ [3]) && \text{using (2)}\\
&= 1 + (2 + (3 + \texttt{mySum}\ [\,])) && \text{using (2)}
\end{aligned}$$

Evaluating Functions by Hand (aka Equational Reasoning)

- functions are defined by equations and pattern matching
- general idea: "replace equals by equals"

Example (mySum)

given the two equations

$$\texttt{mySum [] = 0} \tag{1}$$
$$\texttt{mySum (x:xs) = x + mySum xs} \tag{2}$$

we evaluate `mySum [1,2,3]` like

`mySum [1,2,3]`	`= 1 + mySum [2,3]`	using (2)
	`= 1 + (2 + mySum [3])`	using (2)
	`= 1 + (2 + (3 + mySum []))`	using (2)
	`= 1 + (2 + (3 + 0))`	using (1)

Evaluating Functions by Hand (aka Equational Reasoning)

- functions are defined by equations and pattern matching
- general idea: "replace equals by equals"

Example (mySum)

given the two equations

$$\texttt{mySum}\ [\,] = \texttt{0} \tag{1}$$

$$\texttt{mySum}\ (\texttt{x}:\texttt{xs}) = \texttt{x} + \texttt{mySum}\ \texttt{xs} \tag{2}$$

we evaluate mySum [1,2,3] like

$$\begin{aligned} \texttt{mySum}\ [1,2,3] &= 1 + \texttt{mySum}\ [2,3] && \text{using (2)} \\ &= 1 + (2 + \texttt{mySum}\ [3]) && \text{using (2)} \\ &= 1 + (2 + (3 + \texttt{mySum}\ [\,])) && \text{using (2)} \\ &= 1 + (2 + (3 + 0)) && \text{using (1)} \\ &= 6 && \text{by def. of} + \end{aligned}$$

Outline

Haskell

- is a pure language (only allowing “explicit” side effects)
- functions are defined by equations and pattern matching

Haskell

- is a pure language (only allowing "explicit" side effects)
- functions are defined by equations and pattern matching

Example (quicksort)

- sort list of elements smaller than or equal to x
- sort list of elements larger than x
- insert x in between

```haskell
qsort []     = []
qsort (x:xs) = qsort le ++ [x] ++ qsort gt
  where
    le = [a | a <- xs, a <= x] -- list comprehension
    gt = [b | b <- xs, b > x]
```

Outline

Haskell on the Web

- main entry point `www.haskell.org`
- most widely used Haskell compiler: GHC
- with interpreter GHCi

Haskell on the Web

- main entry point `www.haskell.org`
- most widely used Haskell compiler: GHC
- with interpreter GHCi

Starting the Interpreter (GHCi)

```
$ ghci
GHCi, version 8.2.2: http://www.haskell.org/ghc/
:? for help
...
Prelude>
```

The Standard Prelude

on startup GHCi loads the “Prelude”, importing many standard functions

The Standard Prelude

on startup GHCi loads the "Prelude", importing many standard functions

Examples

- arithmetic: +, -, *, /, ^, `mod`, `div`
- lists

`drop n xs`	drop first n elements from list xs
`head xs`	extract first element from list xs
`length xs`	number of elements in list xs
`product xs`	multiply elements of list xs
`reverse xs`	reverse list xs
`sum xs`	sum up elements of list xs
`tail xs`	obtain list xs without its first element
`take n xs`	take first n elements from list xs

- note: in code examples Prelude functions are colored green and others blue; variables are colored dark orange

Function Application

- in mathematics: function application is denoted by enclosing arguments in parentheses, whereas multiplication of two arguments is often implicit (by juxtaposition)
- in Haskell: reflecting its primary status, function application is denoted silently (by juxtaposition), whereas multiplication is denoted explicitly by *

Function Application

- in mathematics: function application is denoted by enclosing arguments in parentheses, whereas multiplication of two arguments is often implicit (by juxtaposition)
- in Haskell: reflecting its primary status, function application is denoted silently (by juxtaposition), whereas multiplication is denoted explicitly by *

Examples

Mathematics	Haskell
$f(x)$	`f x`
$f(x, y)$	`f x y`
$f(g(x))$	`f (g x)`
$f(x, g(y))$	`f x (g y)`
$f(x)\,g(y)$	`f x * g y`
$f(a, b) + c\,d$	`f a b + c * d`

Haskell Scripts

- define new functions inside scripts
- text file containing definitions
- common suffix `.hs`

Haskell Scripts

- define new functions inside scripts
- text file containing definitions
- common suffix `.hs`

My First Script – `test.hs`

- set editor from inside GHCi `:set editor` **`code`**
- start editor `:edit test.hs` and type

```
double x    = x + x
quadruple x = double (double x)
```

- load script

```
Prelude> :load test.hs
[1 of 1] Compiling Main ( test.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Interpreter Commands

Command	Meaning
`:load` ⟨*filename*⟩	load script ⟨*filename*⟩
`:reload`	reload current script
`:edit` ⟨*filename*⟩	edit script ⟨*filename*⟩
`:edit`	edit current script
`:`**`type`** ⟨*expression*⟩	show type of ⟨*expression*⟩
`:set` ⟨*property*⟩	change various settings
`:show` ⟨*info*⟩	show various information
`:!` ⟨*command*⟩	execute ⟨*command*⟩ in shell
`:?`	show help text
`:quit`	bye-bye!

Example Session

```
> :load test.hs
> quadruple 10
40
> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
> :edit test.hs

factorial n = product [1..n]
average ns  = sum ns `div` length ns

> :reload
> factorial 10
3628800
> average [1,2,3,4,5]
3
```

enclosing function in `...` turns it infix

Naming Requirements

names of functions and their arguments have to conform to following syntax

choice

$$
\begin{aligned}
\langle \mathit{lower} \rangle &::= \texttt{a} \mid \ldots \mid \texttt{z} \\
\langle \mathit{upper} \rangle &::= \texttt{A} \mid \ldots \mid \texttt{Z} \\
\langle \mathit{digit} \rangle &::= \texttt{0} \mid \ldots \mid \texttt{9} \\
\langle \mathit{name} \rangle &::= (\langle \mathit{lower} \rangle \mid \_)(\langle \mathit{lower} \rangle \mid \langle \mathit{upper} \rangle \mid \langle \mathit{digit} \rangle \mid \texttt{'} \mid \_)^{*}
\end{aligned}
$$

Naming Requirements

names of functions and their arguments have to conform to following syntax

choice

zero or more times

$$
\begin{aligned}
\langle \textit{lower} \rangle &::= \texttt{a} \mid \ldots \mid \texttt{z} \\
\langle \textit{upper} \rangle &::= \texttt{A} \mid \ldots \mid \texttt{Z} \\
\langle \textit{digit} \rangle &::= \texttt{0} \mid \ldots \mid \texttt{9} \\
\langle \textit{name} \rangle &::= (\langle \textit{lower} \rangle \mid \_)(\langle \textit{lower} \rangle \mid \langle \textit{upper} \rangle \mid \langle \textit{digit} \rangle \mid \texttt{'} \mid \_)^{*}
\end{aligned}
$$

Naming Requirements

names of functions and their arguments have to conform to following syntax

choice

zero or more times

$$
\begin{aligned}
\langle \mathit{lower} \rangle &::= \mathtt{a} \mid \ldots \mid \mathtt{z} \\
\langle \mathit{upper} \rangle &::= \mathtt{A} \mid \ldots \mid \mathtt{Z} \\
\langle \mathit{digit} \rangle &::= \mathtt{0} \mid \ldots \mid \mathtt{9} \\
\langle \mathit{name} \rangle &::= (\langle \mathit{lower} \rangle \mid \_)(\langle \mathit{lower} \rangle \mid \langle \mathit{upper} \rangle \mid \langle \mathit{digit} \rangle \mid \text{'} \mid \_)^{*}
\end{aligned}
$$

Reserved Names

case class data default deriving do else foreign if import in infix infixl infixr instance let module newtype of then type where _

Naming Requirements

names of functions and their arguments have to conform to following syntax

choice

zero or more times

$$
\begin{aligned}
\langle lower \rangle &::= \texttt{a} \mid \ldots \mid \texttt{z} \\
\langle upper \rangle &::= \texttt{A} \mid \ldots \mid \texttt{Z} \\
\langle digit \rangle &::= \texttt{0} \mid \ldots \mid \texttt{9} \\
\langle name \rangle &::= (\langle lower \rangle \mid \_)(\langle lower \rangle \mid \langle upper \rangle \mid \langle digit \rangle \mid {}' \mid \_)^*
\end{aligned}
$$

Reserved Names

case class data default deriving do else foreign if import in infix infixl infixr instance let module newtype of then type where _

Examples

`myFun fun1 arg_2 x'`

The Layout Rule

- items that start in same column are grouped together
- by increasing indentation, single item may span multiple lines
- groups end at EOF or when indentation decreases
- script content is group, start nested group by **`where`**, **`let`**, **`do`**, or **`of`**
- **ignore layout:** enclose groups in ‘{’ and ‘}’ and separate items by ‘;’

The Layout Rule

- items that start in same column are grouped together
- by increasing indentation, single item may span multiple lines
- groups end at EOF or when indentation decreases
- script content is group, start nested group by **where**, **let**, **do**, or **of**
- **ignore layout:** enclose groups in '{' and '}' and separate items by ';'

Examples

with layout:

```haskell
main =
  let x = 1
      y = 1
  in
  putStrLn (take
    (x+y) (zs++us))
  where
    zs = []
    us = "abc"
```

without layout:

```haskell
main =
  let { x = 1; y = 1 } in
  putStrLn (take (x+y) (zs++us))
  where { zs = []; us = "abc" }
```

Comments

there are two kinds of comments

- single-line comments: starting with `--` and extending to EOL
- multi-line comments: enclosed in `{-` and `-}`

Comments

there are two kinds of comments

- single-line comments: starting with -- and extending to EOL
- multi-line comments: enclosed in {- and -}

Examples

```haskell
-- Factorial of a positive number:
factorial n = product [1..n]

-- Average of a list of numbers:
average ns  = sum ns `div` length ns

{- currently not used
double x    = x + x
quadruple x = double (double x)
-}
```

Thanks! & Questions?