

CS 220: Survey of Programming Languages

LECTURE SLIDES

Functional Programming Languages

Jan Michael C. Yap

Algorithms and Complexity Laboratory
Department of Computer Science
University of the Philippines, Diliman
jcyap@dcs.upd.edu.ph

Session 6



Functional Programming Languages

Functional Programming Languages

LISP

Haskell



Functional Programming Languages

Functional Programming Languages

LISP

Haskell



Church-Turing thesis and Turing-completeness



¹<http://www.cse.uconn.edu/~dqg/papers/cie05.pdf>

Church-Turing thesis and Turing-completeness

- The **Church-Turing thesis**¹: “Whenever there is an **effective method (algorithm) for obtaining the values of a mathematical function**, the function can be computed by a T[uring] M[achine].”



¹<http://www.cse.uconn.edu/~dqg/papers/cie05.pdf>

Church-Turing thesis and Turing-completeness

- The **Church-Turing thesis**¹: “Whenever there is an **effective method (algorithm) for obtaining the values of a mathematical function**, the function can be computed by a T[uring] M[achine].”
- **Turing-completeness** is a criterion of a computational system that can simulate any (single-taped) Turing machine



¹<http://www.cse.uconn.edu/~dqg/papers/cie05.pdf>

Church-Turing thesis and Turing-completeness

- The **Church-Turing thesis**¹: “Whenever there is an **effective method (algorithm) for obtaining the values of a mathematical function**, the function can be computed by a T[uring] M[achine].”
- **Turing-completeness** is a criterion of a computational system that can simulate any (single-taped) Turing machine



¹<http://www.cse.uconn.edu/~dqg/papers/cie05.pdf>

Church-Turing thesis and Turing-completeness

- The **Church-Turing thesis**¹: “Whenever there is an **effective method (algorithm) for obtaining the values of a mathematical function**, the function can be computed by a T[uring] M[achine].”
- **Turing-completeness** is a criterion of a computational system that can simulate any (single-taped) Turing machine
 - Systems here include **computational hardware, model of computation, and programming languages**



¹<http://www.cse.uconn.edu/~dqg/papers/cie05.pdf>

The λ -calculus



The λ -calculus

- Developed by Alonzo Church in the 1930's as a means of **describing computation**, and is equivalent to Turing machines (Turing-complete!)



The λ -calculus

- Developed by Alonzo Church in the 1930's as a means of **describing computation**, and is equivalent to Turing machines (Turing-complete!)
- One of the precursors of **symbolic computation**



The λ -calculus

- Developed by Alonzo Church in the 1930's as a means of **describing computation**, and is equivalent to Turing machines (Turing-complete!)
- One of the precursors of **symbolic computation**
- "... [T]he **smallest universal programming language** of (sic) the world."



The λ -calculus

- Developed by Alonzo Church in the 1930's as a means of **describing computation**, and is equivalent to Turing machines (Turing-complete!)
- One of the precursors of **symbolic computation**
- "... [T]he **smallest universal programming language** of (sic) the world."

$$\langle expr \rangle := \langle name \rangle \mid \langle func \rangle \mid \langle application \rangle \mid (\langle expr \rangle)$$

$$\langle func \rangle := \lambda \langle name \rangle . \langle expr \rangle$$

$$\langle application \rangle := \langle expr \rangle \langle expr \rangle$$


The λ -calculus - Basic Examples

- Identity function: $I \equiv \lambda x.x$



The λ -calculus - Basic Examples

- Identity function: $I \equiv \lambda x.x$
 - $I(y) \equiv (\lambda x.x)y$
 - $I(z) \equiv (\lambda x.x)z$
 - $I(I) \equiv (\lambda x.x)(\lambda z.z)$



The λ -calculus - Basic Examples

- Identity function: $I \equiv \lambda x.x$
 - $I(y) \equiv (\lambda x.x)y$
 - $I(z) \equiv (\lambda x.x)z$
 - $I(I) \equiv (\lambda x.x)(\lambda z.z)$
- Natural number representations



The λ -calculus - Basic Examples

- Identity function: $I \equiv \lambda x.x$
 - $I(y) \equiv (\lambda x.x)y$
 - $I(z) \equiv (\lambda x.x)z$
 - $I(I) \equiv (\lambda x.x)(\lambda z.z)$
- Natural number representations
 - $0 \equiv \lambda s z.z$
 - $1 \equiv \lambda s z.s(z)$
 - $2 \equiv \lambda s z.s(s(z))$
 - $3 \equiv \lambda s z.s(s(s(z)))$



The λ -calculus - Basic Examples

- Identity function: $I \equiv \lambda x.x$
 - $I(y) \equiv (\lambda x.x)y$
 - $I(z) \equiv (\lambda x.x)z$
 - $I(I) \equiv (\lambda x.x)(\lambda z.z)$
- Natural number representations
 - $0 \equiv \lambda s z.z$
 - $1 \equiv \lambda s z.s(z)$
 - $2 \equiv \lambda s z.s(s(z))$
 - $3 \equiv \lambda s z.s(s(s(z)))$
- Successor function: $S \equiv \lambda w y x.y(w y x)$



The λ -calculus - Basic Examples

- Identity function: $I \equiv \lambda x.x$
 - $I(y) \equiv (\lambda x.x)y$
 - $I(z) \equiv (\lambda x.x)z$
 - $I(I) \equiv (\lambda x.x)(\lambda z.z)$
- Natural number representations
 - $0 \equiv \lambda sz.z$
 - $1 \equiv \lambda sz.s(z)$
 - $2 \equiv \lambda sz.s(s(z))$
 - $3 \equiv \lambda sz.s(s(s(z)))$
- Successor function: $S \equiv \lambda wyx.y(wyx)$
 - $S(1) \equiv (\lambda wyx.y(wyx))(\lambda sz.s(z))$
 - $S(3) \equiv (\lambda wyx.y(wyx))(\lambda sz.s(s(s(z))))$
 - $2S(3) \equiv (\lambda sz.s(s(z)))(\lambda wyx.y(wyx))(\lambda ab.a(a(a(b))))$



Characteristics of an functional PL



Characteristics of an functional PL

- Based on the concepts of **mathematical functions**



Characteristics of an functional PL

- Based on the concepts of **mathematical functions**
 - Computation often defined by **separation into cases**



Characteristics of an functional PL

- Based on the concepts of **mathematical functions**
 - Computation often defined by **separation into cases**
- No “variables”, assignments, and iterative constructs



Characteristics of an functional PL

- Based on the concepts of **mathematical functions**
 - Computation often defined by **separation into cases**
- No “variables”, assignments, and iterative constructs
 - There are **identifiers** bound to values, but not in a variable “sense” in that its value can be changed by an assignment statement



Characteristics of an functional PL

- Based on the concepts of **mathematical functions**
 - Computation often defined by **separation into cases**
- No “variables”, assignments, and iterative constructs
 - There are **identifiers** bound to values, but not in a variable “sense” in that its value can be changed by an assignment statement
 - Values are **bound** to identifiers, not assigned. At times, values bound to identifiers are **substituted**



Characteristics of an functional PL

- Based on the concepts of **mathematical functions**
 - Computation often defined by **separation into cases**
- No “variables”, assignments, and iterative constructs
 - There are **identifiers** bound to values, but not in a variable “sense” in that its value can be changed by an assignment statement
 - Values are **bound** to identifiers, not assigned. At times, values bound to identifiers are **substituted**
 - Only means of iteration is through **recursion**



Functional Programming Languages

Functional Programming Languages

LISP

Haskell



Background



Background

- In 1958, John McCarthy drafted a programming language for doing **symbolic computation**, which became the first draft of LISP



Background

- In 1958, John McCarthy drafted a programming language for doing **symbolic computation**, which became the first draft of LISP
- There are two basic data types: **atoms and lists**



Background

- In 1958, John McCarthy drafted a programming language for doing **symbolic computation**, which became the first draft of LISP
- There are two basic data types: **atoms and lists**
 - Atoms are **stand alone literals or identifiers**: 1, e, id



Background

- In 1958, John McCarthy drafted a programming language for doing **symbolic computation**, which became the first draft of LISP
- There are two basic data types: **atoms and lists**
 - Atoms are **stand alone literals or identifiers**: 1, e, id
 - Lists are delimited by **parentheses**: (a b c d), (e), ((f g) h)



Background

- In 1958, John McCarthy drafted a programming language for doing **symbolic computation**, which became the first draft of LISP
- There are two basic data types: **atoms and lists**
 - Atoms are **stand alone literals or identifiers**: 1, e, id
 - Lists are delimited by **parentheses**: (a b c d), (e), ((f g) h)
- An expression/statement is structured such that **an operator/function precedes its operands/parameters** (Polish prefix notation)



Background

- In 1958, John McCarthy drafted a programming language for doing **symbolic computation**, which became the first draft of LISP
- There are two basic data types: **atoms and lists**
 - Atoms are **stand alone literals or identifiers**: 1, e, id
 - Lists are delimited by **parentheses**: (a b c d), (e), ((f g) h)
- An expression/statement is structured such that **an operator/function precedes its operands/parameters** (Polish prefix notation)
 - (+ 1 3)
 - (car H T)



Evaluation

- Data types⁴



⁴<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node15.html>

Evaluation

- Data types⁴
 - Two “primitive” types: atomic types and list types



⁴<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node15.html>

Evaluation

- Data types⁴
 - Two “primitive” types: atomic types and list types
 - Integers, real and complex floating point numbers, and characters as atomic types



⁴<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node15.html>

Evaluation

- Data types⁴
 - Two “primitive” types: atomic types and list types
 - Integers, real and complex floating point numbers, and characters as atomic types
 - List types can include atomic and/or list types



⁴<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node15.html>

Evaluation

- Data types⁴
 - Two “primitive” types: atomic types and list types
 - Integers, real and complex floating point numbers, and characters as atomic types
 - List types can include atomic and/or list types
 - Arrays are supported



⁴<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node15.html>

Evaluation

- Data types⁴
 - Two “primitive” types: atomic types and list types
 - **Integers, real and complex floating point numbers, and characters** as atomic types
 - **List types** can include atomic and/or list types
 - **Arrays** are supported
 - **Functions** are also treated as (LISP) objects

⁴<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node15.html>



Evaluation

- Syntax design (Basic LISP^{5,6}):

⁵<http://cui.unige.ch/isi/bnf/LISP/BNFlisp.html>

⁶<http://ep.yimg.com/ty/cdn/paulgraham/jmc.lisp>

⁷Manually counted

⁸<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node43.html>

⁹<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node111.html>



Evaluation

- Syntax design (Basic LISP^{5,6}):
 - Production rules⁷: 17
 - Number of top alternatives⁷: 10
 - Number of symbols⁷: 37
 - Vocabulary
 - Nonterminal symbols⁷: 11
 - Terminal symbols⁷: 47

⁵<http://cui.unige.ch/isi/bnf/LISP/BNFlisp.html>

⁶<http://ep.yimg.com/ty/cdn/paulgraham/jmc.lisp>

⁷Manually counted

⁸<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node43.html>

⁹<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node111.html>



Evaluation

- Syntax design (Basic LISP^{5,6}):
 - Production rules⁷: 17
 - Number of top alternatives⁷: 10
 - Number of symbols⁷: 37
 - Vocabulary
 - Nonterminal symbols⁷: 11
 - Terminal symbols⁷: 47
 - Very simple, **near λ -calculus specs**

⁵<http://cui.unige.ch/isi/bnf/LISP/BNFlisp.html>

⁶<http://ep.yimg.com/ty/cdn/paulgraham/jmc.lisp>

⁷Manually counted

⁸<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node43.html>

⁹<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node111.html>



Evaluation

- Syntax design (Basic LISP^{5,6}):
 - Production rules⁷: 17
 - Number of top alternatives⁷: 10
 - Number of symbols⁷: 37
 - Vocabulary
 - Nonterminal symbols⁷: 11
 - Terminal symbols⁷: 47
 - Very simple, **near λ -calculus specs**
 - Allows for programs and other functions to be **built using only a handful syntactic components**

⁵<http://cui.unige.ch/isi/bnf/LISP/BNFlisp.html>

⁶<http://ep.yimg.com/ty/cdn/paulgraham/jmc.lisp>

⁷Manually counted

⁸<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node43.html>

⁹<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node111.html>



Evaluation

- Syntax design (Basic LISP^{5,6}):
 - Production rules⁷: 17
 - Number of top alternatives⁷: 10
 - Number of symbols⁷: 37
 - Vocabulary
 - Nonterminal symbols⁷: 11
 - Terminal symbols⁷: 47
 - Very simple, **near λ -calculus specs**
 - Allows for programs and other functions to be **built using only a handful syntactic components**
- Abstraction:

⁵<http://cui.unige.ch/isi/bnf/LISP/BNFlisp.html>

⁶<http://ep.yimg.com/ty/cdn/paulgraham/jmc.lisp>

⁷Manually counted

⁸<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node43.html>

⁹<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node111.html>



Evaluation

- Syntax design (Basic LISP^{5,6}):
 - Production rules⁷: 17
 - Number of top alternatives⁷: 10
 - Number of symbols⁷: 37
 - Vocabulary
 - Nonterminal symbols⁷: 11
 - Terminal symbols⁷: 47
 - Very simple, **near λ -calculus specs**
 - Allows for programs and other functions to be **built using only a handful syntactic components**
- Abstraction:
 - No access modifiers, all defined functions are “public” in the duration it is running/invoked ⁸

⁵<http://cui.unige.ch/isi/bnf/LISP/BNFlisp.html>

⁶<http://ep.yimg.com/ty/cdn/paulgraham/jmc.lisp>

⁷Manually counted

⁸<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node43.html>

⁹<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node111.html>



Evaluation

- Syntax design (Basic LISP^{5,6}):
 - Production rules⁷: 17
 - Number of top alternatives⁷: 10
 - Number of symbols⁷: 37
 - Vocabulary
 - Nonterminal symbols⁷: 11
 - Terminal symbols⁷: 47
 - Very simple, **near λ -calculus specs**
 - Allows for programs and other functions to be **built using only a handful syntactic components**
- Abstraction:
 - No access modifiers, all defined functions are “public” in the duration it is running/invoked ⁸
 - Common Lisp employs a **package system** to form a name space for identifiers used ⁹

⁵<http://cui.unige.ch/isi/bnf/LISP/BNFlisp.html>

⁶<http://ep.yimg.com/ty/cdn/paulgraham/jmc.lisp>

⁷Manually counted

⁸<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node43.html>

⁹<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node111.html>



Evaluation



¹⁰<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node117.html>

¹¹<http://www.gigamonkeys.com/book/beyond-exception-handling-conditions-and-restarts.html>

Evaluation

- Expressivity: Has some built-in packages, but mostly for **basic I/O and for aiding function definitions** ¹⁰



¹⁰<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node117.html>

¹¹<http://www.gigamonkeys.com/book/beyond-exception-handling-conditions-and-restarts.html>

Evaluation

- Expressivity: Has some built-in packages, but mostly for **basic I/O and for aiding function definitions** ¹⁰
- Type checking:



¹⁰<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node117.html>

¹¹<http://www.gigamonkeys.com/book/beyond-exception-handling-conditions-and-restarts.html>

Evaluation

- Expressivity: Has some built-in packages, but mostly for **basic I/O and for aiding function definitions** ¹⁰
- Type checking:
- Exception handling: ¹¹



¹⁰<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node117.html>

¹¹<http://www.gigamonkeys.com/book/beyond-exception-handling-conditions-and-restarts.html>

Evaluation

- Expressivity: Has some built-in packages, but mostly for **basic I/O and for aiding function definitions** ¹⁰
- Type checking:
- Exception handling: ¹¹
 - **define-condition** to define a custom exception/error



¹⁰<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node117.html>

¹¹<http://www.gigamonkeys.com/book/beyond-exception-handling-conditions-and-restarts.html>

Evaluation

- Expressivity: Has some built-in packages, but mostly for **basic I/O and for aiding function definitions** ¹⁰
- Type checking:
- Exception handling: ¹¹
 - `define-condition` to define a custom exception/error
 - `handler-case` to define function in handling a condition (of a particular `condition-type`)

¹⁰<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node117.html>

¹¹<http://www.gigamonkeys.com/book/beyond-exception-handling-conditions-and-restarts.html>



Evaluation

- Expressivity: Has some built-in packages, but mostly for **basic I/O and for aiding function definitions** ¹⁰
- Type checking:
- Exception handling: ¹¹
 - **define-condition** to define a custom exception/error
 - **handler-case** to define function in handling a condition (of a particular **condition-type**)
 - **restart-case** to define function recovering from error and issuing restart of function run



¹⁰<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node117.html>

¹¹<http://www.gigamonkeys.com/book/beyond-exception-handling-conditions-and-restarts.html>

Evaluation

- Restricted aliasing: Initially had pass-by-name, but since deprecated; now only employs **pass-by-value**¹²



¹²<https://www.cl.cam.ac.uk/teaching/1213/ConceptsPL/l4.pdf>
¹³<https://common-lisp.net/project/cmucldoc/cmu-user/compiler-hint.html>
¹⁴<http://c2.com/cgi/wiki?WeakAndStrongTyping>
¹⁵http://www.iaeng.org/IJCS/issues_v32/issue_4/IJCS_32_4_19.pdf

Evaluation

- Restricted aliasing: Initially had pass-by-name, but since deprecated; now only employs **pass-by-value**¹²
- Efficiency^{13,14}



¹²<https://www.cl.cam.ac.uk/teaching/1213/ConceptsPL/l4.pdf>

¹³<https://common-lisp.net/project/cmucl/doc/cmu-user/compiler-hint.html>

¹⁴<http://c2.com/cgi/wiki?WeakAndStrongTyping>

¹⁵http://www.iaeng.org/IJCS/issues_v32/issue_4/IJCS_32_4_19.pdf

Evaluation

- Restricted aliasing: Initially had pass-by-name, but since deprecated; now only employs **pass-by-value**¹²
- Efficiency^{13,14}
 - LISP employs **dynamic** type checking, but nonetheless is **strongly-typed**



¹²<https://www.cl.cam.ac.uk/teaching/1213/ConceptsPL/l4.pdf>

¹³<https://common-lisp.net/project/cmucl/doc/cmu-user/compiler-hint.html>

¹⁴<http://c2.com/cgi/wiki?WeakAndStrongTyping>

¹⁵http://www.iaeng.org/IJCS/issues_v32/issue_4/IJCS_32_4_19.pdf

Evaluation

- Restricted aliasing: Initially had pass-by-name, but since deprecated; now only employs **pass-by-value**¹²
- Efficiency^{13,14}
 - LISP employs **dynamic** type checking, but nonetheless is **strongly-typed**
 - Generally an **interpreted language**, but Common Lisp (at least) allows **code compilation**



¹²<https://www.cl.cam.ac.uk/teaching/1213/ConceptsPL/l4.pdf>

¹³<https://common-lisp.net/project/cmucl/doc/cmu-user/compiler-hint.html>

¹⁴<http://c2.com/cgi/wiki?WeakAndStrongTyping>

¹⁵http://www.iaeng.org/IJCS/issues_v32/issue_4/IJCS_32_4_19.pdf

Evaluation

- Restricted aliasing: Initially had pass-by-name, but since deprecated; now only employs **pass-by-value**¹²
- Efficiency^{13,14}
 - LISP employs **dynamic** type checking, but nonetheless is **strongly-typed**
 - Generally an **interpreted language**, but Common Lisp (at least) allows **code compilation**
 - **Tail recursions** are almost always imminent in coding

¹²<https://www.cl.cam.ac.uk/teaching/1213/ConceptsPL/l4.pdf>

¹³<https://common-lisp.net/project/cmucl/doc/cmu-user/compiler-hint.html>

¹⁴<http://c2.com/cgi/wiki?WeakAndStrongTyping>

¹⁵http://www.iaeng.org/IJCS/issues_v32/issue_4/IJCS_32_4_19.pdf



Evaluation

- Restricted aliasing: Initially had pass-by-name, but since deprecated; now only employs **pass-by-value**¹²
- Efficiency^{13,14}
 - LISP employs **dynamic** type checking, but nonetheless is **strongly-typed**
 - Generally an **interpreted language**, but Common Lisp (at least) allows **code compilation**
 - **Tail recursions** are almost always imminent in coding
 - **cons** is an **expensive function**

¹²<https://www.cl.cam.ac.uk/teaching/1213/ConceptsPL/l4.pdf>

¹³<https://common-lisp.net/project/cmucl/doc/cmu-user/compiler-hint.html>

¹⁴<http://c2.com/cgi/wiki?WeakAndStrongTyping>

¹⁵http://www.iaeng.org/IJCS/issues_v32/issue_4/IJCS_32_4_19.pdf



Evaluation

- Restricted aliasing: Initially had pass-by-name, but since deprecated; now only employs **pass-by-value**¹²
- Efficiency^{13,14}
 - LISP employs **dynamic** type checking, but nonetheless is **strongly-typed**
 - Generally an **interpreted language**, but Common Lisp (at least) allows **code compilation**
 - **Tail recursions** are almost always imminent in coding
 - **cons** is an **expensive function**
 - “The main cause of inefficiency is the **compiler’s lack of adequate information about the types of function argument and result values.**”

¹²<https://www.cl.cam.ac.uk/teaching/1213/ConceptsPL/l4.pdf>

¹³<https://common-lisp.net/project/cmucl/doc/cmu-user/compiler-hint.html>

¹⁴<http://c2.com/cgi/wiki?WeakAndStrongTyping>

¹⁵http://www.iaeng.org/IJCS/issues_v32/issue_4/IJCS_32_4_19.pdf



Evaluation

- Restricted aliasing: Initially had pass-by-name, but since deprecated; now only employs **pass-by-value**¹²
- Efficiency^{13,14}
 - LISP employs **dynamic** type checking, but nonetheless is **strongly-typed**
 - Generally an **interpreted language**, but Common Lisp (at least) allows **code compilation**
 - **Tail recursions** are almost always imminent in coding
 - **cons** is an **expensive function**
 - “The main cause of inefficiency is the **compiler’s lack of adequate information about the types of function argument and result values.**”
 - Can be made to run at par with C programs **with optimized coding**¹⁵

¹²<https://www.cl.cam.ac.uk/teaching/1213/ConceptsPL/l4.pdf>

¹³<https://common-lisp.net/project/cmucl/doc/cmu-user/compiler-hint.html>

¹⁴<http://c2.com/cgi/wiki?WeakAndStrongTyping>

¹⁵http://www.iaeng.org/IJCS/issues_v32/issue_4/IJCS_32_4_19.pdf



Functional Programming Languages

Functional Programming Languages

LISP

Haskell



Background



<http://research.microsoft.com/en-us/um/people/simonpj/papers/history-of-haskell/history.pdf>

Background

- Conceptualized in a meeting between Paul Hudak, Philip Wadler, and Peyton Jones in 1987 .

<http://research.microsoft.com/en-us/um/people/simonpj/papers/history-of-haskell/history.pdf>



Background

- Conceptualized in a meeting between Paul Hudak, Philip Wadler, and Peyton Jones in 1987 .
- Founded on the idea of lazy functional languages, and partially based on the design of the Miranda programming language, but more on an open standard



Background

- Conceptualized in a meeting between **Paul Hudak, Philip Wadler, and Peyton Jones** in 1987 .
- Founded on the idea of **lazy functional languages**, and partially based on the design of the **Miranda** programming language, but more on an **open standard**
- The first version of Haskell was released in April 1990.



Evaluation

- Data types^{16,17}



¹⁶<http://learnyouahaskell.com/starting-out>

¹⁷<http://learnyouahaskell.com/types-and-typeclasses>

¹⁸<https://www.haskell.org/onlinereport/syntax-iso.html>

¹⁹Manually counted

Evaluation

- Data types^{16,17}
 - Integers, real (floating point) numbers, characters, logical as primitive types



¹⁶<http://learnyouahaskell.com/starting-out>

¹⁷<http://learnyouahaskell.com/types-and-typeclasses>

¹⁸<https://www.haskell.org/onlinereport/syntax-iso.html>

¹⁹Manually counted

Evaluation

- Data types^{16,17}
 - Integers, real (floating point) numbers, characters, logical as primitive types
 - Lists in lieu of arrays



¹⁶<http://learnyouahaskell.com/starting-out>

¹⁷<http://learnyouahaskell.com/types-and-typeclasses>

¹⁸<https://www.haskell.org/onlinereport/syntax-iso.html>

¹⁹Manually counted

Evaluation

- Data types^{16,17}
 - Integers, real (floating point) numbers, characters, logical as primitive types
 - Lists in lieu of arrays
- Syntax design (Haskell 98^{18,19})



¹⁶<http://learnyouahaskell.com/starting-out>

¹⁷<http://learnyouahaskell.com/types-and-typeclasses>

¹⁸<https://www.haskell.org/onlinereport/syntax-iso.html>

¹⁹Manually counted

Evaluation

- Data types^{16,17}
 - Integers, real (floating point) numbers, characters, logical as primitive types
 - Lists in lieu of arrays
- Syntax design (Haskell 98^{18,19})
 - Production rules: 224
 - Number of top alternatives: 194
 - Number of symbols: 92
 - Vocabulary
 - Nonterminal symbols: 69
 - Terminal symbols: 147

¹⁶<http://learnyouahaskell.com/starting-out>

¹⁷<http://learnyouahaskell.com/types-and-typeclasses>

¹⁸<https://www.haskell.org/onlinereport/syntax-iso.html>

¹⁹Manually counted



Evaluation

- Abstraction²⁰:



²⁰<http://pharo.gforge.inria.fr/PBE1/PBE1ch6.html>

²¹<http://learnyouahaskell.com/modules>

²²<http://book.realworldhaskell.org/read/error-handling.html>

Evaluation

- Abstraction²⁰:
 - Methods are **public** and instance variables are **private** (to the instances that own it)



²⁰<http://pharo.gforge.inria.fr/PBE1/PBE1ch6.html>

²¹<http://learnyouahaskell.com/modules>

²²<http://book.realworldhaskell.org/read/error-handling.html>

Evaluation

- Abstraction²⁰:
 - Methods are **public** and instance variables are **private** (to the instances that own it)
 - **Classes are not in a namespace**, so all class names must be unique.



²⁰<http://pharo.gforge.inria.fr/PBE1/PBE1ch6.html>

²¹<http://learnyouahaskell.com/modules>

²²<http://book.realworldhaskell.org/read/error-handling.html>

Evaluation

- Abstraction²⁰:
 - Methods are **public** and instance variables are **private** (to the instances that own it)
 - **Classes are not in a namespace**, so all class names must be unique.
- Expressivity: Has a rich set of modules containing basic (predefined) functions ²¹



²⁰<http://pharo.gforge.inria.fr/PBE1/PBE1ch6.html>

²¹<http://learnyouahaskell.com/modules>

²²<http://book.realworldhaskell.org/read/error-handling.html>

Evaluation

- Abstraction²⁰:
 - Methods are **public** and instance variables are **private** (to the instances that own it)
 - **Classes are not in a namespace**, so all class names must be unique.
- Expressivity: Has a rich set of modules containing basic (predefined) functions ²¹
- Exception handling: Has **Maybe** and **Either** keywords for conditional based exception handling, define custom errors well as the **Control.Exception** module ²²



²⁰<http://pharo.gforge.inria.fr/PBE1/PBE1ch6.html>

²¹<http://learnyouahaskell.com/modules>

²²<http://book.realworldhaskell.org/read/error-handling.html>

Evaluation

- Restricted aliasing: Purely **pass-by-value**^{23,24}

²³<http://web.cecs.pdx.edu/~harry/compilers/slides/ParamPassing.pdf>

²⁴<http://courses.cs.washington.edu/courses/cse341/04wi/lectures/22-parameter-passing.html>

²⁵<http://learnyouahaskell.com/types-and-typeclasses>

²⁶https://wiki.haskell.org/Haskell_programming_tips

²⁷<http://users.aber.ac.uk/afc/stricthaskell.html>



Evaluation

- Restricted aliasing: Purely **pass-by-value**^{23,24}
- Efficiency



²³<http://web.cecs.pdx.edu/~harry/compilers/slides/ParamPassing.pdf>

²⁴<http://courses.cs.washington.edu/courses/cse341/04wi/lectures/22-parameter-passing.html>

²⁵<http://learnyouahaskell.com/types-and-typeclasses>

²⁶https://wiki.haskell.org/Haskell_programming_tips

²⁷<http://users.aber.ac.uk/afc/stricthaskell.html>

Evaluation

- Restricted aliasing: Purely **pass-by-value**^{23,24}
- Efficiency
 - Haskell employs **static** type checking.²⁵



²³<http://web.cecs.pdx.edu/~harry/compilers/slides/ParamPassing.pdf>

²⁴<http://courses.cs.washington.edu/courses/cse341/04wi/lectures/22-parameter-passing.html>

²⁵<http://learnyouahaskell.com/types-and-typeclasses>

²⁶https://wiki.haskell.org/Haskell_programming_tips

²⁷<http://users.aber.ac.uk/afc/stricthaskell.html>

Evaluation

- Restricted aliasing: Purely **pass-by-value**^{23,24}
- Efficiency
 - Haskell employs **static** type checking.²⁵
 - As with most functional PLs, Haskell also suffers from inefficiency due to **lazy evaluation**, but nonetheless efficiency can be attained through **coding optimization**^{26,27}

²³<http://web.cecs.pdx.edu/~harry/compilers/slides/ParamPassing.pdf>

²⁴<http://courses.cs.washington.edu/courses/cse341/04wi/lectures/22-parameter-passing.html>

²⁵<http://learnyouahaskell.com/types-and-typeclasses>

²⁶https://wiki.haskell.org/Haskell_programming_tips

²⁷<http://users.aber.ac.uk/afc/stricthaskell.html>



END OF SESSION 6

