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
  - Systems here include computional hardware, model of computation, and programming languages

[1] http://www.cse.uconn.edu/∼dqg/papers/cie05.pdf

The λ-calculus

http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf

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!)

http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf

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

http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf

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."

http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf

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."

$$
\begin{aligned}
< expr > &:= < name > \mid < func > \mid < application > \mid (< expr >) \\
< func > &:= \lambda < name > . < expr > \\
< application > &:= < expr >< expr >
\end{aligned}
$$

http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf

The λ-calculus - Basic Examples

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

http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf

The λ-calculus - Basic Examples

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

http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf

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

http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf

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
  - $0 \equiv \lambda sz.z$
  - $1 \equiv \lambda sz.s(z)$
  - $2 \equiv \lambda sz.s(s(z))$
  - $3 \equiv \lambda sz.s(s(s(z)))$

---

http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf

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
  - $0 \equiv \lambda sz.z$
  - $1 \equiv \lambda sz.s(z)$
  - $2 \equiv \lambda sz.s(s(z))$
  - $3 \equiv \lambda sz.s(s(s(z)))$
- Succesor function:$S \equiv \lambda wyx.y(wyx)$

---

http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf

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
  - $0 \equiv \lambda sz.z$
  - $1 \equiv \lambda sz.s(z)$
  - $2 \equiv \lambda sz.s(s(z))$
  - $3 \equiv \lambda sz.s(s(s(z)))$
- Succesor function:$S \equiv \lambda wyx.y(wyx)$
  - $S(1) \equiv (\lambda wyx.y(wyx))(\lambda sz.s(z))$
  - $S(3) \equiv (\lambda wyx.y(wyx))(\lambda sz.s(s(s(z))))$
  - $2S(3) \equiv (\lambda sz.s(s(z)))(\lambda wyx.y(wyx))(\lambda ab.a(a(a(b))))$

http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf

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

http://groups.engin.umd.umich.edu/CIS/course.des/cis400/lisp/lisp.html

Background

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

http://groups.engin.umd.umich.edu/CIS/course.des/cis400/lisp/lisp.html

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

http://groups.engin.umd.umich.edu/CIS/course.des/cis400/lisp/lisp.html

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)

---

http://groups.engin.umd.umich.edu/CIS/course.des/cis400/lisp/lisp.html

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)

http://groups.engin.umd.umich.edu/CIS/course.des/cis400/lisp/lisp.html

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)

---

http://groups.engin.umd.umich.edu/CIS/course.des/cis400/lisp/lisp.html

Evaluation

- Data types[4]

[4] https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node15.html

Evaluation

- Data types[4]
    - Two "primitive" types: atomic types and list types

[4] https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node15.html

Evaluation

- Data types[4]
    - 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 types[4]
  - 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 types[4]
  - 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 types[4]
  - 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
[7] Manually 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 rules[7]: 17
  - Number of top alternatives[7]: 10
  - Number of symbols[7]: 37
  - Vocabulary
    - Nonterminal symbols[7]: 11
    - Terminal symbols[7]: 47

[5] http://cui.unige.ch/isi/bnf/LISP/BNFlisp.html
[6] http://ep.yimg.com/ty/cdn/paulgraham/jmc.lisp
[7] Manually 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 rules[7]: 17
  - Number of top alternatives[7]: 10
  - Number of symbols[7]: 37
  - Vocabulary
    - Nonterminal symbols[7]: 11
    - Terminal symbols[7]: 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
[7] Manually 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 rules[7]: 17
  - Number of top alternatives[7]: 10
  - Number of symbols[7]: 37
  - Vocabulary
    - Nonterminal symbols[7]: 11
    - Terminal symbols[7]: 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
[7] Manually 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 rules[7]: 17
  - Number of top alternatives[7]: 10
  - Number of symbols[7]: 37
  - Vocabulary
    - Nonterminal symbols[7]: 11
    - Terminal symbols[7]: 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
[7] Manually 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 rules[7]: 17
  - Number of top alternatives[7]: 10
  - Number of symbols[7]: 37
  - Vocabulary
    - Nonterminal symbols[7]: 11
    - Terminal symbols[7]: 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
[7] Manually 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 rules[7]: 17
  - Number of top alternatives[7]: 10
  - Number of symbols[7]: 37
  - Vocabulary
    - Nonterminal symbols[7]: 11
    - Terminal symbols[7]: 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
[7] Manually 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]
- 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]
- 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]
- 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]
- 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/cmucl/doc/cmu-user/compiler-hint.html
[14] http://c2.com/cgi/wiki?WeakAndStrongTyping
[15] http://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
[15] http://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
[15] http://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
[15] http://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
[15] http://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
[15] http://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
[15] http://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
[15] http://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

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
- The first version of Haskell was released in April 1990.

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

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
[19] Manually 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
[19] Manually counted

Evaluation

- Data types[16,17]
  - Integers, real (floating point) numbers, characters, logical as primitive types
  - Lists in lieu of arrays, tuples as immutable lists, and strings as derived types

[16] http://learnyouahaskell.com/starting-out
[17] http://learnyouahaskell.com/types-and-typeclasses
[18] https://www.haskell.org/onlinereport/syntax-iso.html
[19] Manually counted

Evaluation

- Data types[16,17]
  - Integers, real (floating point) numbers, characters, logical as primitive types
  - Lists in lieu of arrays, tuples as immutable lists, and strings as derived types
- 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
[19] Manually counted

Evaluation

- Data types[16,17]
  - Integers, real (floating point) numbers, characters, logical as primitive types
  - Lists in lieu of arrays, tuples as immutable lists, and strings as derived types
- 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
[19] Manually counted

Evaluation

- Abstraction[20]:

[20] http://pharo.gforge.inria.fr/PBE1/PBE1ch6.html
[21] http://learnyouahaskell.com/modules
[22] http://book.realworldhaskell.org/read/error-handling.html

Evaluation

- Abstraction[20]:
  - 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

- Abstraction[20]:
  - 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

- Abstraction[20]:
  - 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

- Abstraction[20]:
  - 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
[26] https://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
26 https://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
[26] https://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
[26] https://wiki.haskell.org/Haskell_programming_tips
[27] http://users.aber.ac.uk/afc/stricthaskell.html

END OF SESSION 6