1. History (including ancestors and descendants)
   1. Ancestors
      1. ML (Ref 1, pg 707)
      2. Miranda
   2. Descendants
      1. Ocaml (Ref 1, pg 712)
      2. F# (Ref 1, ppg 712)
2. Design goals
   1. Initially wanted to base the language on Miranda. (Ref 4, pg 12-4)
   2. We wanted a language that could be used, among other purposes, for research into language features (Ref 4, pg 12-4)
   3. We sought the freedom for anyone to extend or modify the language (Ref 4, pg 12-4)
   4. Sought to build and distribute an implementation. (Ref 4, pg 12-4)
   5. The committee was asked by David Turner (Miranda creator) to be sufficiently different from Miranda. (Ref 4, pg 12-4)
   6. It should be suitable for teaching, research, and applications, including building large systems. (Ref 4, pg 12-4)
   7. It should be completely described via the publication of a formal syntax and semantics. (Ref 4, pg 12-4)
   8. It should be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please. (Ref 4, pg 12-4)
   9. It should be usable as a basis for further language research. (Ref 4, pg 12-4)
   10. It should be based on ideas that enjoy a wide consensus. (Ref 4, pg 12-4)
   11. It should reduce unnecessary diversity in functional programming languages. More specifically, we initially agreed to base it on an existing language, namely OL. (Ref 4, pg 12-4)
   12. The last two goals reflected the fact that we intended the language to be quite conservative, rather than to break new ground. Although matters turned out rather differently, we intended to do little more than embody the current consensus of ideas and to unite our disparate groups behind a single design.
   13. As we shall see, not all of these goals were realised. We abandoned the idea of basing Haskell explicitly on OL very early.
   14. We violated the goal of embodying only well-tried ideas, notably by the inclusion of type classes
   15. We never developed a formal semantics.
3. Syntax ( including a complete syntax description if available in the appendix)
4. Data types, type checking and scoping
   1. Data Types
      1. Basic Types
         1. Boolean (Bool) (Ref 5, pg 33) (Ref 6, pg 21)
            1. True/False
            2. Does not coerce integers or other types to boolean. (Ref 6, pg 5)
            3. Literal values which do not need to be evaluated, as the evaluation returns the literal itself (Ref 5, pg 34)
            4. Operators

&& (and) (C-influenced) (Ref 6, pg 5)

|| (or) (C-influenced) (Ref 6, pg 5)

not (Ref 6, pg 6)

== (equal to)

/= (not equal to) (Ref 6, pg 6)

* + - 1. Integer (Int or Integer) (Ref 5, pg 35) (Ref 6, pg 21)
         1. Int type represents integers in a fixed amount of space, and so can only represent a finite range of integers.

The value maxBound gives the greatest value in the type (2147483647)

* + - * 1. Integer type may accurately represent whole numbers of any size as long as it fits into memory.
        2. Operators (Ref 5, pg 36)

Arithmetic

+ (sum of two integers)

\* (product of two integers

^ (raise to the power)

Infix: 2^3 → 8

- (difference of two integers)

Infix: a-b

Prefix (negative): -a

Haskell's only unary operator which should use parentheses to avoid confusing the compiler when used with the other infix operators (Ref 6, pg 4)

div (whole number division)

Prefix: div 14 3 → 4

Infix: 14 `div` 3 → 4 (must include backticks; any function can be implemented in this way)

mod (remainder from whole number division

Prefix: mod 14 3 → 2

Infix: 14 `mod` 3 → 2 (must include backticks)

abs (absolute value of an integer; remove the sign)

negate (changes the sign of an integer)

Relational (return a Bool) (Ref 5, pg 36)

> (greater than [and not equal to])

>= (greater than or equal to)

== (equal to)

/= (not equal to)

<= (less than or equal to

< (less than [and not equal to])

Guards?

* + - 1. Characters (Char)
         1. Literal characters represented inside single quotes (Ref 5, pg 41)
         2. Special characters (Ref 5, pg 42)

'\t' (tab)

'\n' (newline)

'\\' (backslash)

'\'' (single quote)

'\”' (double quote)

* + - * 1. Uses Unicode (Ref 5, pg 42)
        2. Conversion functions

ord (get the numerical code of a character)

chr (get the character representation of a numerical code)

* + - 1. Numbers with fractional parts(Ref 5, pg 43-44)
         1. Floating-point numbers (Float)

Operators

+ - \* (add, subtract, multiply)

/ (Fractional division

^ (exponentiation x^n = xn for a natural number *n*)

\*\* (exponentiation x\*\*y = xy) (allows for decimal numbers in the exponent)

==, /=, <, >, <=, >= (Equality and ordering opens)

abs (absolute value)

acos, asin, atan (the inverse of cosine, sine, and tangent)

ceiling, floor, round (convert a fraction to an integer by rounding up, down, or to the closest integer)

cos, sin, tan (cosine, sine, and tangent)

exp (powers of e)

fromInt (convert an Int to a Float)

log (logarithm to base e)

logBase (logarithm to arbitrary base, provided as first argument)

negate (change the sign of a number

pi (the constant pi)

signum (1.0, 0.0, or -1.0 according to whether the argument is positive, zero, or negative)

sqrt ([Positive] square root)

* + - * 1. Double (Double; greater precision than Float)
        2. Rational (Rational; full precision numbers built from Integer)

Operators

% (Numerator on left, denominator on right) (Ref 6, pg 14)

Must load in +Data.Ratio module

Only handles type Ratio Integer

* + - * 1. Scientific notation

Example: 231.61e7 → 231.61 x 107 = 2,316,100,000

* + 1. Other Types
       1. Tuples (Ref 5, pg 72)
          1. Built from other data types

Strings

Integers

List of Doubles

Tuple of Tuples

* + - * 1. Type is denoted by parentheses and each tuple has its own specific type

(data type)

* + - * 1. May contain different values

(String, Integer)

* + - * 1. May contain any arbitrary number of elements

There is not a one-element tuple (Ref 6, pg 25)

* + - * 1. Special tuple (Ref 6, pg 25)

Empty

()

Type and value are usually referred to as a “unit” (this may be for all tuples?)

Similar to void in C

* + - * 1. Pattern matching over functions? (Ref 5, pg 74-75)
      1. Lists (Ref 5, pg 73)
         1. Built from data types

Characters

Integers

Tuples of (String, Double)

List of Lists

* + - * 1. Type is denoted by brackets and each list has its own specific type

[Integer]

* + - * 1. May only contain the same types

[(String, Integer), (String, Integer), (String, Integer)]

[Integer, Integer, Integer]

* + - * 1. An empty list is considered a member of a list

[]

* + - * 1. May contain an arbitrary number of elements
        2. May use enumeration (..) on allowed types

Creates a list where content is filled in according to the specified beginning and end of the enumeration

[1..5] → [1,2,3,4,5]

Enumerations may also specify a value to step by

[1.2, 1.4 .. 2.0] → [1.2, 1.4, 1.6, 1.8, 2.0]

If

* + - * 1. Operations

++ (list concatenation) (Ref 6, pg 11)

: (“cons” – adds to head of list) (Ref 6, pg 11)

First argument should be a single element of the type stored in the list, and the second argument should be the actual list.

== (equivalence check)

* + - 1. String (Ref 5, pg 73)
         1. A list of characters

[Char]

Special characters/strings (Ref 6, pg 11)

'\n'

'\t'

More in appendix B

“”S (empty string) is a synonym for [] (Ref 6, pg 12)

Operators

putStrLen (prints a string) (Ref 6 pg 12)

== (equivalence) (Ref 6, pg 12)

++ (concatenate) (Ref 6, pg 12)

: (“cons”) (may be performed on a character as first argument, since strings are lists of characters)

* 1. Algebraic types (Ref 5, pg 242)
  2. Abstract Data type (Ref 5, pg 299)
  3. Type Checking (Ref 5, pg 227)
     1. Monomorphic type checking (Ref 5, pg 228)
        1. This is type checking without polymorphism or overloading
        2. Haskell is not monomorphic (Ref 5, pg 230)
     2. Polymorphic type checking (Ref 5, pg 230)
        1. Haskell is polymorphic (Ref 5 pg 230)
     3. Haskell uses three main aspects to allow abstraction of how we think about types (Ref 6, pg 16-17)
        1. Strong type system (Ref 6, pg 18)
           1. To be *strong* means that the type system guarantees that a program cannot contain type errors.
           2. Type errors occurs when a program does not follow a language's type rules.

Well-typed means that it follows the language's rules

Ill-typed means that it does not follow the language's rules

* + - * 1. Types cannot be coerced
        2. This causes a slight decrease in performance
        3. Catches bugs in a program long before they ever may occur
      1. Static type system (Ref 6, pg 19)
         1. The compiler knows
  1. Polymorphism (Ref 1, pg 422)
     1. Ad hoc polymorphism
        1. Type classes in Haskell
     2. Subtype polymorphism
        1. Does not exist
     3. Parametric polymorphism
        1. Abstract data types

1. Data structures
   1. Association Lists (Ref 6, pg 299)
      1. A list containing (key, value) tuples
      2. Can be built like any other list
      3. Built-in function
         1. Data.List.lookup looks up data in an association list
            1. Has type Eq a => a → [(a,b)] → Maybe b
   2. Map type provided by Data.Map module (Ref 6, pg 299)
      1. Similar in functionality to association lists (Ref 6, pg 301)
      2. Has considerable performance advantage over association lists.
      3. Gives same capabilities as hash tables in other languages.
      4. Is implemented internally as a balanced binary tree
      5. Functions (Ref 6, pg 302)
         1. A map may be constructed by intially building an association list, and then passing it as an argument to a constructor
            1. mapFromAL = Map.fromList al
         2. A map may be createde by using a fold on an association list
            1. mapFold = foldl (\map (k,v) → Map.insert k v map) Map.empty al
         3. Map.insert key value manually returns a value that may be assigned
            1. These statements may be chained together with a “.” and the following statement may be on the next line. The last line must have “$ Map.empty” at the end
2. Control Structures
   1. If/Else
      1. Both branches have to return the same type
   2. Monads?
   3. Continuations (maybe, maybe not)
      1. A current program's state (look up on wikipedia)
   4. Guards
      1. | (pipe)
      2. Otherwise
   5. Case expressions
3. Subprograms
4. Data Abstraction
5. Parameter passing
6. Concurrency
   1. Built-in, happens in run-time...magic....
7. Recursion (Ref 5, pg 53)
   1. Tail call optimization
8. Exception Handling
   1. Pure
   2. Monadic
      1. Easiest way to do handle exceptions
      2. Allows changes to the control flow
9. Expressions and the Assignment Statement
10. Input/Output
    1. Done with Monads
       1. In a pure a language, a function with a given input should return the same result. This means that reading a line should not return a different result. This is why we need states.
       2. State Monad
11. Other unusual features
12. Contributions to the programming language landscape
13. A sample program
14. Global Issues
15. Promise for the future.
16. GHCI Interpreter
17. Turing Machine vs Lambda Calculus
    1. Turing machine
    2. Lambda calculus as it relates to Haskell
       1. Haskell is based on Lambda calculus
    3. Both are effective models of computation
    4. Lambda is more trackable
18. Type classes
    1. A way to look up a piece of code by type
       1. Example:
          1. class Eq a where

(==), (/=) :: a → a → Bool

* + - 1. These two things are defined as boolean for these two operators of the class
      2. We can later define an instance with an operator as:
         1. intstance Eq () where

() == () = True

1. Monads
   1. A container type that can put together two containers
   2. Must follow three laws
   3. A module
2. Functors

Works Cited

1. Sebesta, Robert W. “Concepts of Programming Languages.” 10th Ed.

2. <https://www.haskell.org/pipermail/haskell/2009-November/021750.html> – Changes from Haskell 98 to Haskell 2010

3. “A History of Haskell: Being Lazy With Class” (ACM)

4. Conception, Evolution, and Application of Functional Programming Languages (ACM)

5. Thompson, Simon. “The Craft of Functional Programming” (Book)

6. “Real World Haskell” (Book)

Notes:

1. Come back to write about the interpreter (ghci) (Ref 6, ch. 1)

2. Determine where to write about operator precedence (Ref 6, pg 7)

3. Determine where to talk about Undefined values and Haskell' standard library (Prelude) (Ref 6, pg 8)