The HEX Programming Language Specification

Draft v0.1

Yanzheng Li

Initiated Jan 25, 2012

Table of Contents

The HEX Programming Language Specification
Draft v0.1

Table of Contents
1.0 Introduction
    1.1 Licensing
2.0 Language Overview
    2.1 Overview
    2.2 Design Philosophies
        2.2.1 Simplicity & Readability
        2.2.2 Versatility & Scalability
    2.3 Data Typing
        2.3.1 Optional Typing
        2.3.2 Static Typing
        2.3.3 Dynamic Typing
    2.4 Scope
    2.5 Basic Data Types and Structures
        2.5.1 Basic Data Types
        2.5.2 Basic Data Structures
    2.6 Indentation
        2.6.1 Line Structure
        2.6.2 Explicit line joining
        2.6.3 Implicit Line Joining
        2.6.4 Indentation
    2.7 Comments
        2.7.1 Comment in HEX
        2.7.2 Inline Comment
        2.7.2 Multi-line Comment
3.0 Variables
    3.1 Variable Declaration
        3.1.1 Const Variable
    3.2 Memory Management
        3.2.1 Stack Variables
        3.2.2 Strong Reference
        3.2.3 Weak Reference
    3.3 Lazy Initialization & Deferred Evaluation
        3.1 Lazy Initialization
        3.2 Deferred Evaluation
4.0 Functions
    4.1 Static Functions
    4.2 Function Parameters
        4.2.1. Default Parameters
        4.2.2 Variadic Parameters
    4.3 Function Signature and Overloading
    4.4 Parameter Passing
        4.4.1 Passing by Value
        4.4.2 Passing by Reference
5.0 Expression
    5.1 Expressions
    5.2 Expression Value
    5.3 Literals
    5.4 Invocation Expression
    5.5 Query Expression
    5.6 Lambda Expression

5.6.1 Lambda Expression
5.6.2 Indentation
5.6.3 Variable Passing & Scope
5.6.4 Lambda For Class Member
6.0 Statements
6.1 For Statement
6.2 If Statement
6.3 While Statement
6.4 Try-catch statement
7.0 Operators
7.1 Operators
7.1.1 Arithmetic
7.1.2 Assignment
7.1.3 Augmented Assignment
7.1.4 Bitwise assignment
7.1.5 Augmented bitwise assignment
7.1.6 Bitwise shift
7.1.7 Augmented bitwise shift
7.1.8 Boolean logic
7.1.9 Conditional evaluation
7.2 Operator Precedence
8.0 Struct
8.1 Struct Definition
8.2 Anonymous Struct
9.0 Class
9.1 Class Definition
9.1 Constructor
9.1.1 Default Constructor
9.1.2 Constructor Overloading
9.3 Class Members
9.4 Access Specifiers
9.5 Class Operator and Operator Overload
9.6 Extension Method
9.7.1 Private members for extension methods
10.0 Interface
10.1 Interface Definition
11.0 Task
12.0 Compiler Properties and Attributes
12.1 Compiler Property
12.2. Attributes

1.0 Introduction

1.1 Licensing

2.0 Language Overview

2.1 Overview

HEX is a multi-paradigm programming language that supports both static and dynamic types, and was designed with the core principles of simplicity, readability, versatility and scalability to allow developers to create a diversity of types of computer programs with modern language features, succinct syntax and semantics that are built into the core of the language construct.

2.2 Design Philosophies

2.2.1 Simplicity & Readability

HEX's syntax and semantics resembles a lot of the modern programming languages that you've probably heard of or even used. For example, HEX has a indented syntax very much like Python. It supports object-oriented programming with class definitions very much like C++ and Java. It utilizes basic control statements such as if, elif, and loop types such as for, while just like pretty much every other languages out there. However, HEX does have its own syntax and semantics for aspects of the language that might not seem familiar to you, which you will discover as you read along this document.

HEX was designed to offer a minimal learning curve for people to pick up its syntax and semantics by keeping conventions that people are already accustomed to, yet offering a distinct set of new features and capabilities to developers.

2.2.2 Versatility & Scalability

HEX is a general purpose programming language that its usage is not tied to a specific area of development, and its diverse spectrum of features and paradigms supports both high-level and low-level programming tasks. Code written in HEX can be transitioned easily from testing prototypes to production-level code.

2.2.3 Modularity and the HEX Standard Library

The core of the HEX programming language is designed to be simple and lightweight. The core language only supports the runtime and the core types in the language.

The HEX standard library is the default library that is shipped as a side component for the language, it contains a vast spectrum of utilities for basic operations such as string manipulations, file I/O, networking, etc.

2.3 Data Typing

2.3.1 Optional Typing

HEX supports both static and dynamics typed objects, and the two can be mixed together in the same code. While dynamic types offer the simplicity of code and efficiency, static types

allows applications to be safer, modular and complex, and supports high-level features such as generics and interfaces. The use of either one or both types of typing mode depends on the personal preference and the nature of the application.

2.3.2 Static Typing

Static type annotations can be used in variable declaration, member variable declarations, function parameters and return types. Whenever static typed objects are used, type checking is enforced, both during the compilation process and during runtime. If there is a static type mismatch during compilation, the compiler will issue an error and compilation aborts. During runtime, whenever an expression generates a type mismatch, the HEX runtime will issue an error and aborts the program.

2.3.3 Dynamic Typing

Local and global variables, member variables, function parameters and return types can be be used as dynamic typed objects. A dynamically typed object can have its type changed from any expressions anytime during runtime. Declarations of these objects do not require the static type annotation. Using the `typeof` keyword to determine the type of a dynamically typed object. For a dynamically typed object that has been declared but not instantiated or assigned to will yield the type of “unknown”.

2.4 Scope

HEX is lexically scoped and defines scoping for functions and type declarations both at the module level as well as local level. The compiler will issue a warning if there is more than one declaration of either a type, function and variable with the same name at their respective scope. When a variable declaration in an inner scope has the same name as another variable declaration in the outer scope, the inner scope hides the out scope and is used when referred.

2.5 Basic Data Types and Structures

2.5.1 Basic Data Types

Table 1 below consists all the built-in data types supported by HEX.

Type	Size(bit)	Range	Note
`bool`	8	0 or 1	

char	8	-128 to 127	
uchar	8	0 to 255	
short	16	-32768 to 32767	
ushort	16	0 to 65536	
int	32	-2147483648 to 2147483647	
uint	32	0 to 4294967296	
long	64	-9223372036854775808 to 9223372036854775807	
ulong	64	0 to 18446744073709551615	
float	32	+/- 1.4023x10-45 to 3.4028x10+38	general purpose real number
double	64	+/- 4.9406x10-324 to 1.7977x10308	high precision real number

2.5.2 Basic Data Structures

Table 2 below consists all the built-in data types supported by HEX.

Type	Literal Example	Description
string	"this is a string"	Immutable null-terminated string.
array	int numbers[5] = {1,2,3,4,5}	Immutable array
list	["HEX", 5, 'e']	Mutable array-type list
tuple	("HEX", 5, 'e')	Immutable tuple
struct	{name = "HEX", version = 0.1}	Mutable struct
set	(["HEX", 5, 'e'])	Unordered collection of unique data
map	{"apple" : "sweet", "orange" : "sour"}	Key-value dictionary

`multimap`	`{“apple” : (“sweet”, “red”), “orange” : (”sour”, “orange”)}`	Single key multiple value dictionary

2.6 Indentation

2.6.1 Line Structure

HEX source code is divided into logic lines. A logical line is marked by the token NEWLINE at the end. A logic line is consisted of one or more physical lines by following explicit and implicit line joining rules.

2.6.2 Explicit line joining

Two or more physical lines may be joined together into logical lines using backslash characters(\).

When a physical line ends with a backslash that is not part of a string literal or comment, it is joined with the following forming a single logical line.

A line ending with a backslash cannot carry a comment. A backslash does not continue a comment. A backslash does not continue a token except for string literals. In other words, tokens other than string literals cannot be split across physical lines using a backslash. A backslash is illegal elsewhere on a line outside a string literal.

2.6.3 Implicit Line Joining

Expressions in parentheses, square brackets, or curly braces can be split over more than one physical line without using backslashes.

For example:

```
// 5 x 5 2D array
int numbers[][] = {{1,2,3,4,5},
                   {6,7,8,9,0},
                   {11,12,13,14,15},
                   {16,17,18,19,20},
                   {21,22,23,24,25}}
```

2.6.4 Indentation

Leading whitespace(spaces) at the beginning of a logical line is used to compute the indentation level, which is in turn used to determine the grouping of statements.

The total number of spaces preceding the first non-blank character determines the line's indentation. Indentation cannot be split over multiple physical lines using backslashes.

The indentation levels of consecutive lines are used to generate INDENT and DEDENT tokens using a stack:

NOTE: HEX v0.1, INDENT is restricted to double spaces only.

2.7 Comments

2.7.1 Comment in HEX

HEX supports both inline and multi-line comments. The comment style for both type is the same as the ones in common languages such as C, C++ and Java.

2.7.2 Inline Comment

An inline comment starts with `//` and ends at the end of the line

e.g.

```
// This is an inline comment
```

2.7.2 Multi-line Comment

A multi-line comment starts with `/*` and ends with `*/` and spans more than one line. Anything in between is considered part of the comment.

e.g.

```
/*
```

```
This is a multi-line comment.
This is a multi-line comment.
This is a multi-line comment.
*/
```

3.0 Variables

3.1 Variable Declaration

Variables of both statically and dynamically types can be declared at both the module level and local level. Statically typed variables require static type annotation before the variable name, whereas dynamically typed variables do not. However, dynamically typed variable declarations must follow either an assignment to a literal value, a reference to another object or its own initialization.

Examples:

```
// Dynamic typed variable declared and assigned to null.
name = null

// Dynamic-types variable declared and instantiated to type string.
name = new string("123")

// Typed variable declaration without initialization and assignment.
string name

// Typed variable declared and assigned string literal 123.
string name = "123"
```

3.1.1 Const Variable

All variables declared must be initially instantiated or assigned to a given value or null. A variable declared with the keyword `const` cannot have its value changed after instantiation or assignment.

```
// Typed variable declared and assigned string literal 123.
const string name = "HEX"

name = "not HEX" // compiler error
```

3.2 Memory Management

All objects declared are allocated on the heap. The HEX runtime has built-in garbage collection mechanisms that periodically cleans up memory that are not longer referenced. Under normal

circumstances, developers do not have to worry about memory management issues and clean up.

The HEX garbage collection uses a reference count scheme to dynamically determine objects need to be deallocated at runtime. The garbage collector frees up memory used by strong references when the execution cannot reach the objects, it also claims memory used by weak references of objects when no strong references are associated with them.

3.2.1 Stack Variables

Objects declared with the `stackalloc` keyword are allocated on the stack, and their lifetime is not regulated by the garbage collector. Member variables cannot be initiated on the stack.

```
// An object of type string is declared and initialized on the stack.
string name = stackalloc new string("Yanzheng Li")
```

For module level objects that are allocated on the stack, they are freed upon program termination, for local level objects, they are freed when their scope ends.

3.2.2 Strong Reference

A strong reference is a variable that references an object in memory at runtime, which cannot be freed up by the garbage collector when it can be reached during execution. All objects initiated or instantiated are strong references by default.

3.2.3 Weak Reference

A weak reference is a variable that references an object in memory at runtime, which can be freed up by the garbage collector when no strong references are associated with it at runtime, even when it still can be reached by code during execution. A weak reference of an object can be obtained by the `~=` assignment operator.

e.g.

```
MyObject weakObj ~= obj
```

A strong reference of a weak reference can be obtained to prevent it from claimed by the garbage collector. However, there is no guarantee that a strong reference will be generated for a weak reference before garbage collection takes place.

3.3 Lazy Initialization & Deferred Evaluation

Sometime, due to the large amount of computation operations, we want to defer evaluations

of expressions and functions and initializations of objects until we need them to improve performance.

3.1 Lazy Initialization

When a object type is large, initialization takes a lot of time and consumes a lot of computing power and resources, but it may not be right away used. Thus, we want a mechanism to delay the actual initialization of the object until it is ever used. This is called lazy initialization, and can be used to improve performance.

To instruct an object to be lazily initialized, use the `lazy` keyword before the new keyword during initialization. Until the lazily initialized object is used, its constructor will not be invoked.

```
// Lazily initializes an object.
// At this time, the object is seem
// to be initialized to the program,
// but its constructor has not
// been invoked yet.
obj = lazy new VeryLazyObject()

// do some work

/* The first time the object is used,
   its constructor is invoked now. */
obj.doSomething()
```

3.2 Deferred Evaluation

The idea of deferred evaluation is similar to that of lazy initialization. Sometime when an object is assigned a value that is the return type of a function or method, it might be not ideal to process that invocation right away because it might be computationally expensive, thus, the invocation can be delayed when that object is ever used. Use the `defer` keyword right before a function invocation to defer its execution

```
// An object declared and assigned to null
obj = null

// Defer some very expensive operation
// The function is not invoked yet
```

```
obj = defer someVeryExpensiveOperation()

// Now that the object is used in a statement
// the function
// someVeryExpensiveOperation() is processed
print obj.value
```

4.0 Functions

4.1 Static Functions

The `static` keyword can be applied to a function to have a special meaning, there are two cases:

1. When applied to a module level function, it means that the function is only visible and accessible within that module. Access to the function from outside of that module will trigger a compiler error.
2. When applied to a class member function(method), it means that the method is global to all instances of that class.

Example:

```
/* Static function, only visible within module. */
def static CalculateFactorial(n):
   return if n < 1 then 1 else CalculateFactorial(n-1)
```

4.2 Function Parameters

Function declarations can have either statically or dynamically typed parameters or a mixture of both. To avoid function declaration ambiguity, all static typed parameters must be declared before all dynamic typed parameters, if any.

```
/* Function declaration and definition */
def CalculateFactorial(n):
   return if n < 1 then 1 else CalculateFactorial(n-1)

/* Function with typed argument(s) */
def CalculateFactorial(int n):
   return if n < 1 then 1 else CalculateFactorial(n-1)

/* Function declaration only. */
def CalculateFactorial(n)
```

4.2.1. Default Parameters

Both static and dynamic typed parameters can have a default value. A default value is assigned to that parameter when the function is invoked.

```
/* Function default parameter. */
def CalculateFactorial(n=0)
def CalculateFactorial(n, recursiveLimit=1000)
```

4.2.2 Variadic Parameters

A function's parameter list can have an indefinite number of parameters, called variadic parameters. Variadic parameters are declared with "…" and must be declared after any static and dynamic typed parameters.

To access all variadic parameters, iterate through `self.args`.

```
/* Function declaration with variadic parameters. */
def CalculateFactorial(n, ...):
    for arg in self.args:
        print arg
```

4.3 Function Signature and Overloading

All functions must have a signature. The signature of a function consists of the function name, and all the parameters with their respective types. Functions here include function declarations and class methods(including constructor and destructor), and class operators.

- Signature for a function or a method:
    - Consists of the name of the function or method and the types and kind(value or reference) of each of its parameters in the order of left to right.
- Signature for a class constructor and destructor
    - Consists of the name of the function or method and the types and kind(value or reference) of each of its parameters in the order of left to right.
- Signature for an operator
    - Consists of the name of the function or method and the types and kind(value or reference) of each of its parameters in the order of left to right, regardless of their types.

Functions with the same signature can be overloaded. The HEX runtime matches function invocations to declarations that match the same signature.

However, rules below define a set of constraints that prevents function declaration ambiguities. Any one of the rules will cause a compilation error if not followed.

- All statically-typed parameters must go before dynamically-typed parameters. Otherwise, the following set of declarations will cause an ambiguity:

```
def doSomething(int a, b, string c)
def doSomething(int a, int b, c)
```

- Overloading functions with minimally the same set of statically-typed parameters in the same order will cause an ambiguity and thus is not allowed, for example:

```
def doSomething(int a, float b, c)
def doSomething(int a, float b, int d)
```

  Both declarations of `doSomething` here as minimally the same set of statically-typed parameters: `int a, float b`, but the declarations are ambiguous.

4.4 Parameter Passing

4.4.1 Passing by Value

Passing by value means that a copy of the original argument is passed to the function instead of the argument itself. Thus, changes to the parameter inside the function will not cause the value of the original argument to change.

- Passing Value Types by Value:

  A value type simply contains its data instead of pointing to the data it contains, thus, passing value types by value do not change their original values.

```
// passing value types by value:
def square(n):
   n *= n
   print("n inside square is: %d", n)


int n = 5
print("n is %d\n", n)
square(n)
print("n is %d\n", n)
```

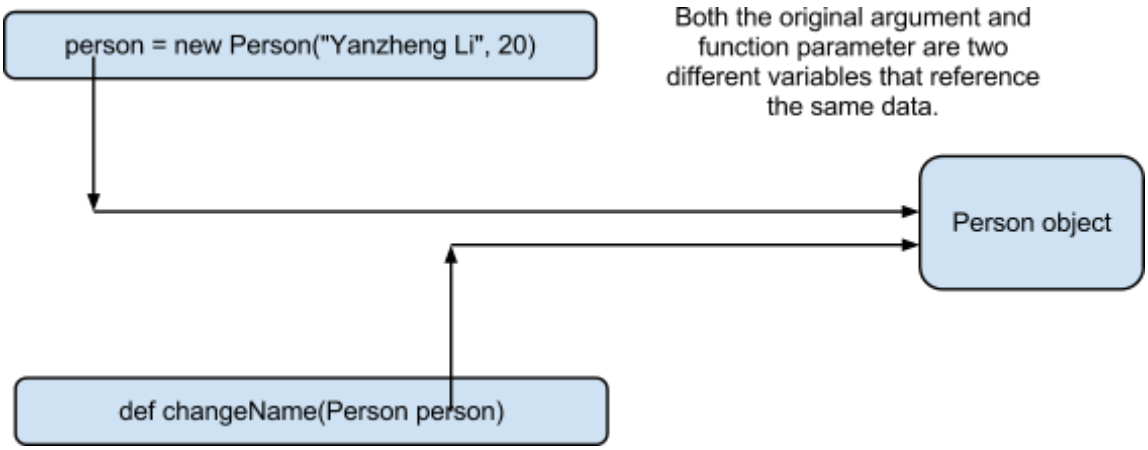
```
// output
// n is 5
// n inside square is: 25
// n is 5
```

- Passing Reference Types by Value:

  When a reference type parameter is passed to a function, the parameter is actually a different reference type object that references the same data. Thus, changes to the data that is referenced by the parameter will be reflected on the original argument.

  For example:

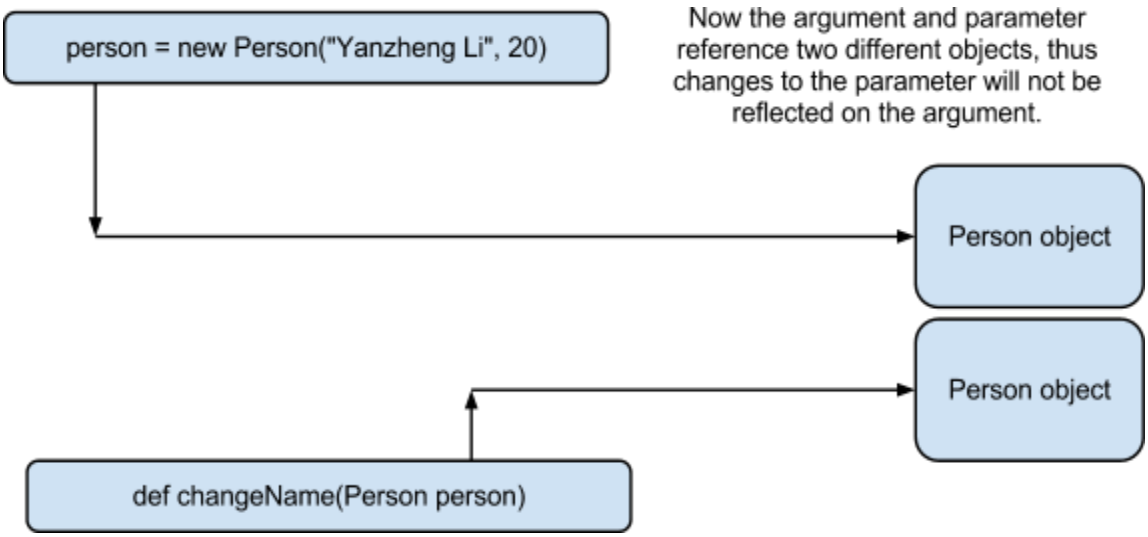
```
def changeName(Person person):
   person.name = "William Li"
   print ("person's name is changed to %s\n", person.name)


person = new Person("Yanzheng Li", 20)
print ("person's name is %s\n", person.name)
changeName(person)
print ("person's name is now %s\n", person.name)

// output
// person's name is Yanzheng Li.
// person's name is changed to William Li.
// person's name is now William Li.
```



  However, the following code will have a different effect.

```
def changeName(Person person):
   person = new Person("William", person.age)
   print("person's name is changed to %s\n", person.name)
```

```
person = new Person(“Yanzheng Li”, 20)
print(“person’s name is %s\n”, person.name)
changeName(person)
print(“person’s name is now %s\n”, person.name)

// output
// person’s name is Yanzheng Li.
// person’s name is changed to William Li.
// person’s name is now Yanzheng Li.
```



4.4.2 Passing by Reference

Passing by reference means that the original argument is passed to the function, instead a copy of itself. Thus, changes to the parameter will be reflected on the argument.

A parameter that is passed by reference is annotated with the `ref` keyword:

- Passing value types by reference:

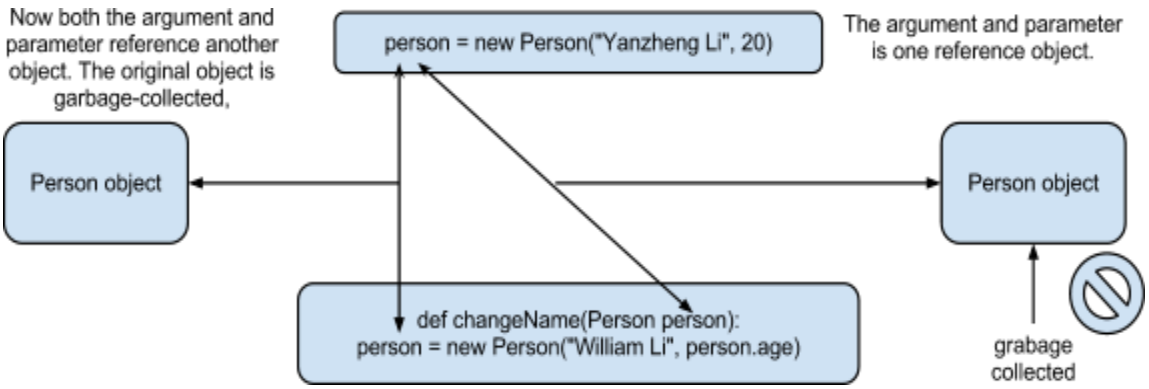
  For example:

```
// passing value types by value:
def square(ref n):
   n *= n
   print(“n inside square is: %d”, n)


int n = 5
```

```
print ("n is %d\n", n)
square(n)
print ("n is %d\n", n)

// output
// n is 5
// n inside square is: 25
// n is 25
```

- Passing reference types by reference:

```
def changeName(ref Person person):
   person = new Person("William", person.age)
   print ("person's name is changed to %s\n", person.name)


person = new Person("Yanzheng Li", 20)
print("person's name is %s\n", person.name)
changeName(person)
print("person's name is now %s\n", person.name)

// output
// person's name is Yanzheng Li.
// person's name is changed to William Li.
// person's name is now William Li.
```



5.0 Expression

5.1 Expressions

An expression is one or more lines of code that is to be evaluated at runtime to yield a value, object or function. Expressions can be consisted of combinations of one or more literal value, method invocation, operators with operands.

5.2 Expression Value

Most expressions are meant to evaluate to a single value, object or function pointer at runtime, with its own associated type.

5.3 Literals

The simplest form of expression is a literal, which is a constant value of any type with no name. This includes string literals, numeric literals, Boolean literals and null.

5.4 Invocation Expression

An invocation expression is a call to either a function, class method or lambda expression.

5.5 Query Expression

Query expression represents a block of code that consists a group of querying calls imperatively over a data source. A query expressions consists of one or more declarative clauses.

A query expressions can operates over a list, map and any data structure that implements the `IEnumerable<T>` interface.

A query expression must begin with the `from` clause and must end with a `select` or `group` clause. In between, the expression can contain one or more clauses of the following types: `where`, `orderby`, `join`, `let`, and additional `from` clauses.

```
// Here is an example of filtering out a list of candidates for an interview.
```

```
experienceGroups = [
                   from applicant in applicants
                   let experience= applicant.education + applicant.previousJobs
                   group applicant by experience into experienceGroup
                   where experience >= 5
                   orderby experience.key()
                   select experienceGroup
                    ]

// Print out applicant's names by experience
for experienceGroup in experienceGroups:
      print("Applicants with experience %u years:\n", experienceGroup.key())
      for applicant in experienceGroup:
            print("\tCandidate: %s\n")
```

5.6 Lambda Expression

5.6.1 Lambda Expression

Lambda expression represents an anonymous function that contains expressions or statements, and can be defined at both module level and function level scope, and can be assigned to a variable to be used for invocation, passed to functions and methods, and attached to events.

All lambda expressions use the lambda operator =>. The left operands are the input parameters that are passed to the lambda expression. At the right side of the operator are blocks of expressions and statements that are to be evaluated and executed when the lambda expression is invoked.

The => operator has the same precedence as the assignment operator = and is right-associative.

For example

```
absolute = (x) => if x < 0 return -1* x

int i = -5
print("i is %d\n", i)
i = absolute(i)
print("absolute value of i is: %d\n", i)

// output
// i is: 5
// absolute value of i is: -5
```

5.6.2 Indentation

When a lambda expression is declared and assigned to a variable, the block of expressions and statements at the right side of the => operator has to follow the same rules of indentation as of any other code that exists elsewhere.

5.6.3 Variable Passing & Scope

Lambda expressions can refer to both outer-scope(declared and defined outside the lambda expression) and inner-scope(declared inside the lambda expression) variables. The following rules dictate variable scope for lambda expressions:

- Variables declared inside a lambda expression are only visible within the expression and are garbage collected when the variable(delegate) that is assigned to this expression is garbage collected.
- A lambda expression cannot reach code outside of the expression itself: e.g., cannot contain a `break` or `continue` statement whose target is outside of the expression itself. Violations of this rule will cause compiler errors.

Lambda expressions following the same rules of variable passing as functions do. By default all variables are passed by value, and need to use the keyword `ref` to pass by reference.

5.6.4 Lambda For Class Member

Lambda expressions can also be used for class methods.

Use the `as` keyword to refer to class members inside the lambda expression.

```
class SomeObject
   public:
      def doSomething() = (x as this.member) => ( return x*x )
```

6.0 Statements

6.1 For Statement

The `for` statement is used to iterate through a list of elements.

Example:

```
for student in students:
  print(student.average)
```

You can also use the `where` keyword to filter out certain elements in the list based on some criteria.

Example:

```
// print out all the students' averages who passed the course
for student in students where student.average >= 50:
  print(student.average)
```

6.2 If Statement

The `if` statement can be used to set conditions for a block of code. Use `elif` to generate one or more alternate conditions, and finally use `else` to capture the default action.

Example:

```
if(student.average >= 80)
     student.rank = excellent
elif(student.average >= 70)
     student.rank = good
elif(student.average >= 50)
     student.rank = satisfactory
else
     student.rank = unsatisfactory
```

6.3 While Statement

The while statement encapsulates a block of code to be executed while a given condition remains true.

Example:

```
int i = 0
while(i <= 100)
  print("%d", i)
  i++
```

6.4 Try-catch statement

A try-catch statement encapsulates a block of code that can catch one or more exceptions

being thrown within that block of code.

Code inside the try statement can throw any number of exceptions. Code inside catch statements following the try statement will handle different types of exceptions being thrown. Last, an optional finally statement can be placed at the end to handle all kinds of situations no matter what happens inside the try statement.

Example:

```
try:
     print("inside try statement...\n")
     throw new Exception("generic exception")
catch:
     print("exception caught")
finally:
    print("gracefully handles the exception")
```

7.0 Operators

7.1 Operators

7.1.1 Arithmetic

addition	subtraction	multiplication	division	modulus
+	-	*	/	%

7.1.2 Assignment

assignment
=

7.1.3 Augmented Assignment

addition	subtraction	multiplication	division	modulus
`+=`	`-=`	`*=`	`/=`	`%=`

7.1.4 Bitwise assignment

negation	bitwise AND	bitwise OR	bitwise XOR
`~`	`&`	`|`	`^`

7.1.5 Augmented bitwise assignment

negation	bitwise AND	bitwise OR	bitwise XOR
`~=`	`&=`	`|=`	`^=`

7.1.6 Bitwise shift

left shift	right shift
`<<`	`>>`

7.1.7 Augmented bitwise shift

left shift	right shift
`<<=`	`>>=`

7.1.8 Boolean logic

negation	logic AND	logic OR
`not`	`and`	`or`

7.1.9 Conditional evaluation

if	else
`if`	`else`

7.1.10 Equality

equality	inequality	less than	less or equal	greater than	equal or

					greater than
==	not	<	<=	>	>=

7.1.11 Increment and decrement

increment	decrement
++	--

7.1.12 Index Operator

index
[]

7.1.13 Lambda Operator

lambda
=>

7.2 Operator Precedence

When multiple operators occur in the same expression, the precedence of the operators dictate the order of which they are evaluated. Table 3 below lists all operators in descending order of precedence.

Table 3. Operator Precedence

Category	Operators
Primary	a.b f(a) a[i] a++ a-- new typeof
Unary	! ~ ++a --a (T)a
Multiplicative	* / %
Additive	+ -
Bitwise Shift	<< >>
Relational	< > <= >=
Equality	== != is not
Bitwise AND	&

Bitwise XOR	`^`
Bitwise OR	`|`
Logic AND	`&&`
Logic OR	`||`
Conditional	`if … then … else …`
Assignment	`= ~= *= /= %= += -= <<= >>= &= ^= |=`

8.0 Struct

8.1 Struct Definition

A struct is a data structure that encapsulates a set of data members of its own of any type, defined by the `struct` keyword. However, it cannot contain any methods that operate on its data members. All members of a struct are publicly accessible.

e.g.

```
struct Person:
   string name
   int age
   hobbies
```

8.2 Anonymous Struct

Sometimes it is necessary to temporarily define a small data structure that only contains a few data members, and a full definition of such a structure is either unnecessary and unmaintainable. Anonymous structs solve this problem by avoid the definition of its members and can still be used within code.

```
// Print out the squares of all even numbers between 1 to 10
data = []
def squareEvenNumbers():
   for i in 1...10 where !(i % 2):
```

```
      data = new { number = i, square = i * i }

for d in data:
   print("%d's square is: %d\n", d.number, d.square)


// output
// 2's square is: 4
// 4's square is: 16
// 6's square is: 36
// 8's square is: 64
// 10's square is: 100
```

9.0 Class

9.1 Class Definition

A class abstractly defines and encapsulates a set of data members and their collective behaviours.

```
class Person : IComparable
    def __init__(name, age):
        this.name = name
        this.age =age
        base.name = name

    def __del__()

    public:
        [#getter=yes, #setter=no]  // compiler property
        string name

        int IsJunior():
            return this.age <= 18

        def virtual IsSenior() = (x as this.age) => x > 65

        def static Print():
            print "This is Person.Print()\n"

    private:
        bool isSenior
```

A class encapsulates a set of member variables and member function(called methods),

constructor, destructor and static members.

9.1 Constructor

A constructor is a unique member function with a unique signature. It is declared with the `def __init__` member function. A constructor serves as a piece of code that get executed when an instance of that class is instantiated with the `new` keyword. Code inside a constructor usually is used to initialize member variables of that instance, and/or execute initialization and validation code to prepare for the creation of an instance. Once all the code gets executed inside the constructor, the object is successfully instantiated. If, however, exceptions are thrown and without being caught inside the constructor will fail to instantiate the object.

9.1.1 Default Constructor

If no custom constructor is defined, the runtime treats the class has an empty constructor.

9.1.2 Constructor Overloading

Constructors overloading follows the same rule as function overloading. At compile time, an object of a particular type upon instantiation is checked against all its constructors, an error is issued when no constructor signature defined inside the class matches the invoking constructor.

9.2 Destructor

A destructor is a unique member function with a unique signature of `__del__()`. It serves as a piece of code that is usually used for clean up purposes. It is invoked when an heap-based object is freed by the garbage collector, or invoked when a stack-based object is freed at the end of its scope. Only one destructor can be declared for a particular class.

9.3 Class Members

There are two types of class members for any class: instance and static members. Instances members are unique members associated with every instance of that class, while static members are only associated with the class itself. Static members does not include constructor and destructor, and is marked with the `static` keyword.

All class members, both instance and static members, must be defined within the same class definition.

9.4 Access Specifiers

There are three sets types of access specifiers that apply to all types of class members: public, protected, and private.

- Public
  The `public` specifier, defined with the keyword public, is used to mark an member that is accessible to both the internal of the class definition itself, all deriving types and outside of the class definition.

- Protected:
  The `protected` specifier, defined with the keyword protected, is used to mark an member that is accessible to both the internal of the class definition itself, and all deriving types, but not to the outside of the class definition.

- Private:
  The `private` specifier, defined with the keyword private: is used to mark an member that is accessible to only the internal of the class definition itself, but not to any of the deriving types and not to the outside of the class definition.

9.5 Class Operator and Operator Overload

Operators can be defined for any classes.

```
def Complex operator+=(Complex c):
  this.real += c.real
  this.imaginary += c.imaginary
  return this
```

9.6 Extension Method

Extension methods are member functions that are added to a type without the need of deriving that particular type.

To apply an extension method to a type, use the `ext` keyword inside the function declaration. The first parameter to the function is the type which the extension method is applied.

e.g.

```
def ext PrintComplex(Complex c):
  print("real: %d\n", c.real)
  print("imaginary: %d\n", c.imaginary)
```

9.7.1 Private members for extension methods

By default, access to private members of a class are forbidden from inside an extension method. However, the extension attribute can be applied to any private members to override this behaviour.

e.g.

```
class Complex:
  public:
    def __init__():
       this.real = 0
       this.imaginary = 0

  private:
    [@ext=yes]     // attribute
    int real
    int imaginary
def ext PrintComplex(Complex c):
  print("real: %d\n", c.real)
  print("imaginary: %d\n", c.imaginary)
```

10.0 Interface

10.1 Interface Definition

An interface defines a set of methods that must be implemented by all classes that implements it. An interface can have any number of methods, getters, setters, and instance variables under any access specifiers.

A compiler error will be issued if any derived type of a particular interface does not have any members specified by that interface,

e.g.

```
// Defines an interface that enforces comparison function.
interface IComparable:
  public:
    bool compare(IComparable c)

// Implements IComparable interface to have comparison function.
class Complex : IComparable
  ...
```

11.0 Task

A task encapsulates a set of actions that is to be performed on its own thread and execution state, with its public members provide its states to the outside world. A task follows all the attributes of a class, except it is defined with the keyword `task`.

The main difference between an object and a task is that a task must have declare and define a member method of signature `void main()` of access type of either protected or private. The following illustrates the definition of a task.

```
task Download:
  public:
    def __init__(name, id, url):
      this.taskName = name
      this.taskID = id
      this.url = url

  protected:
    def main():
     print("Task name: %s", this.taskName)
      print("Task ID: %s", this.taskID)
```

Upon the creation of a task, the order of constructor and destructor invocation following any of its super or base constructors is the same as of an object, and they are invoked on the creating thread.

After constructor invocation, the task is executed by invoking its main() member automatically, and it terminates when its main() member finishes execution. Its destructor is invoked when the block that has that task declared terminates.

Since the main() member of a task is enforced to be private, the task can communicate with the outside world by using its public members, which can exchange information with the execution code and the rest of the code.

12.0 Compiler Properties and Attributes

12.1 Compiler Property

A compiler property is an annotation in the code that instructs the HEX compiler to perform certain actions on the code declared right below the annotation itself.

Compiler properties can be used to enforce clean and concise code by reducing the amount of boilerplate code that developers have to write.

A compiler property has the form of *#property_name=property* value, and a compiler property annotation can have multiple compiler properties listed together, each separated by a comma.

The example below illustrates the use of compiler property to generate a pair of getters and setters for a class member.

```
[#getter=yes, #setter=yes]
string name
```

12.2. Attributes

Attributes are also annotation in the code that specify certain behaviours of the code that are declared right below the annotation itself. However, unlike compiler properties that happen at compile time, attributes annotate behaviours that happen at runtime.

Attributes are useful when additional specific features are desired in the code without modifications on existing code.

An attribute has the form of *@expression*, and an attribute annotation can have multiple attributes listed together, each separated by a comma.

The example below illustrates the use of attribute to specify a for loop to run with parallelism enabled, and validation code on each element in the list.

```
[@for_parallel=yes, @ValidateElement(element)]
for element in elements:
  doCrazyComputation(element)
```