The HEX Programming Language Specification

Draft v0.1

Yanzheng Li
Initiated Jan 25, 2012

Table of Contents

1.0 Introduction

1.1 Licencing

1.2 Notation

2.0 Overview

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

2.1 Design Philosophies

2.1.1 Simplicity & Readability

HEX's syntax and semantics resembles a lot of the modern programming languages that you've probably heard or even used. For example, HEX has a indented syntax very much like Python. It supports objected-oriented programming with class and struct definitions very much like C++ and Java. It utilizes basic control statements such as if, else if, switch, and loop types such as for, while and do 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 seen 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 small set of new features to offer new features and functionalities to developers.

2.1.2 Versatility & Scalability

2.2 Core Features

2.2.1 Optional Typing

HEX supports both static and dynamics types of 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 features such as generics and interfaces. The used either one or both depends on the personal preference and the nature of the application.

2.2.2 Modularity and Core 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.

Additional functionalities for the core value types as well as basic functionalities and utilities for string manipulation, file IO, networking, etc are supported by the core library.

2.3 Scope

HEX is lexically scoped and defines scoping for functions and type declarations are the module level, and at both the module, type definition and function level for variables. 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.

3.0 Variables

3.1 Variable Declaration

Variable declarations must follow assignment, either given null, an instantiated value or reference another variable.

```
// 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 declared and assgined string literal 123.
string name = “123”
```

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.

3.2 Memory Allocation

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. However, objects declared with the `stackalloc` keyword are allocated on the stack, and their lifetime is not regulated by the garbage collector. For module level objects that are allocated on the stack, they are freed upon program termination, for function level objects, they are freed when their scope ends.

3.3 Type Checking

During compilation, the HEX compiler checks for type compatibility for statically-typed objects. Variables that are declared with static types that are not instantiated or assigned with compatible types will cause the compiler to issue an error and thus aborts compilation.

During runtime, the HEX runtime checks type compatibility for all object assignments and instantiations that are statically-typed. Incompatible types may cause program to abort.

4.0 Functions

All functions must have a signature and a body. The signature at least defines the arguments of the function. Functions here include function declarations and class methods(including constructor and destructor).

```
/* Function declaration and definition */
void* CalculateFactorial(n):
   return n < 1 ? 1 : CalculateFactorial(n-1)

/* Function with typed argument(s) */
void* CalculateFactorial(int n):
   return n < 1 ? 1 : CalculateFactorial(n-1)

/* Function with typed return value. */
int CalculateFactorial(n):
   return n < 1 ? 1 : CalculateFactorial(n-1)

/* Function with both typed argument(s) and return value. */
int CalculateFactorial(int n):
   return n < 1 ? 1 : CalculateFactorial(n-1)

/* Static function, only visible within module. */
static void* CalculateFactorial(n):
   return n < 1 ? 1 : CalculateFactorial(n-1)
```

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

/* Function declaration with empty body. */
void* CalculateFactorial(n):
    pass

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

4.1 Function Parameters

Function declarations can have either statically or dynamically typed parameters or a mixture of both.

4.2 Function Signature and Overloading

The following defines the signature for a function, class method, constructor and operator.

- 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.
    - Does not include the return type of the function/method.
- Signature for a class constructor
    - 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 void doSomething(int a, b, string c)
def void doSomething(int a, int b, c)
```

- Overloading functions with minimally the same set of statically-typed parameters in the same order and minimally the same number of dynamically-typed parameters will cause an ambiguity:

  ```
  def void doSomething(int a, float b, c)
  def void doSomething(int a, float b, c, d=0)
  ```

  Both declarations of doSomething here as minimally the same set of statically-typed parameters: int a, float b; and minimally the same number of dynamically-typed p parameters: c.

- df

If none of the function signatures match the signature of that function during invocation, the compiler will issue a warning.

5.0 Classes

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

```
class Person : IComparable
    void __init__(name, age):
        this.name(name)
        this.age(age)
        this.isSenior = age >= 60
       base.name(name)

    void __del__():
```

```
        pass

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

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

    virtual void* IsSenior() = (this.age) => this.age > 60

    static DoNothing():
        pass

private:
    bool isSenior
```

A class encapsulates a set of member variables and member function(called methods), constructor, destructor and static members.

5.1 Constructor

A constructor is a unique member function with a unique signature. It is declared with the `void __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.

5.1.1 Default Constructor

If no custom constructor is defined, the runtime treats the class has a contructor of the following form:

```
void __init__():
  pass
```

5.1.2 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.

5.2 Destructor

A destructor is a unique member function with a unique signature of `void __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.

5.3 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.

5.4 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.

5.5 Operator overload

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

5.6 Inheritance and Polymorphism

A class can inherit one or more interfaces and other classes. The parent class/interface is referred to using the `base` keyword in the deriving class.

Inheritance
Polymorph

5.7 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.

e.g.

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

or

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

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:
    void __init__():
       this.real = 0
       this.imaginary = 0

  private:
    [#ext=yes]
    int real
    int imaginary

Complex.ext.Normalize = (Complex c) => \
  t = sqr(c.real*c.real + c.imaginary*c.imaginary)
  c.real = c.real / t
  c.imaginary = c.imaginary / t
```

6.0 Interface

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
 pass
```

7.0 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 mutual exclusion. 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:
    void __init__():
      this.taskName = "new task"
      this.taskID = 12345

    void __del__():
      pass

  protected:
    void 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.

8.0 Coroutine

A coroutine is a routine that can be suspended at some point and resumed from that particular point when control returns.

```
coroutine Fibonacci:
  private:
    int fn

    void main():
      int fn1, fn2
      fn = 0
      fn1 = fn
      suspend()
```

```
            fn = 1
            fn2 = fn1
            fn1 = fn
            suspend()
            for(;;):
               fn = fn1 + fn2
               fn2 = fn1
               fn1 = fn
               suspend()
    public:
        int next():
            resume()
            return fn
```

9.0 Operators

Arithmetic

addition	subtraction	multiplication	division	modulus
+	-	*	/	%

Assignment

assignment
=

Augmented Assignment

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

Bitwise assignment

negation	logic AND	logic OR	logic XOR
~	&	|	^

Augmented bitwise assignment

negation	logic AND	logic OR	logic XOR
~	&=	|=	^=

Bitwise shift

left shift	right shift
<<	>>

Augmented bitwise shift

left shift	right shift
<<	>>

Boolean logic

negation	logic AND	logic OR
!	&&	||

Conditional evaluation

?	:

Equality

equality	inequality	less than	less or equal	greater than	equal or greater than
`==`	`&&`	`<`	`<=`	`>`	`>=`

Increment and decrement

increment	decrement
`++`	`--`

Index Operator

index
`[]`

Expression Statements

if statement

An if statement defines a conditional execution of a block of code.

```
if 'expression':
    'statement'
```

```
if 'expression':
   'statement'
else if 'expression'
   'statement'
else
   'statement'
```

for statement

```
for 'variable' in 'list':
    'statement'
```

e.g.

```
for egg in eggs:
  FryEgg(egg)
```

```
for i in range(1,1000):
   print("%d\n", i)
```

while statement

```
while 'expression':
    'statement'
```

e.g.

```
while hungry():
  eat()
```

do statement

```
do
    'statement'
```

```
while ‘expression’
```

e.g.

```
i = 5
do
    CalculateFactorial(i)
    i--
while i > 0
```

switch statement

```
switch ‘expression’:
    case ‘switchCase’:
        ‘statement’
    break
    default:
        ‘statement’
     break
```

Basic Data Types

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

Type	Alias	Size(bit)	Range	Note
`bool`		8	0 or 1	
`char`		8	-128 to 127	
`unsigned char`	uchar	8	0 to 255	
`short`		16	-32768 to 32767	
`unsigned short`	ushort	16	0 to 65536	
`int`		32	-2147483648 to 2147483647	
`unsigned int`	uint	32	0 to 4294967296	
`long`		64	-9223372036854775808 to 9223372036854775807	
`unsigned long`	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