The Learn Programing Language

Andres Carrillo

Introduction

Learn:

- general purpose.
- object oriented.
- imperative.
- Compiled
- Statically Typed.
- No Nested Routines.
- Functions are allowed side effects.
- References & Pointer Types.

Goals:

Like the name implies Learn is a language focused on helping new programmers learn how to code. While also reinforcing a general understanding of the underlying structure of computer memory. With these goals in mind Learn is focused on readability, flexibility and the belief that those help reinforce a low cost language.

Focus:

- **Readability:**
  - To reinforce Learn's readability we made it explicitly typed.
    - **Explicit typing:** When declared all variables must declare its type. The following grammar expresses Learn's basic declaration scheme.

      <var_declaration> → <type_name> <identifier> [<assignment>] <end_line>

Focus:

- **Flexibility:** In order to keep with the readability of Learn we pared down its overall constructs. But provided the key tools needed to build user defined data types, and structures.

  <derived_type> → “type” <identifier> < type_name> <end_line>

- In keep with object oriented design Learn comes with polymorphism to allow users to build on top of their custom objects. Both through the use on embedded class/structures and through the defination of interfaces.

Focus

- **Cost:** We believe that focusing on both readability and flexibility naturally lowers the overall cost of Learn as a language.
  - Increasing the readability lowers the cost of a language by making it easy to pick and understand, thereby lowering the overall cost of training.
  - Providing a high degree of flexibility that doesn't muddle readability also helps to lower the cost, by making overall language use easier.

Reserved words:

Reserved Words:

for if else new delete slice array map

continue switch case default return struct func break

interface char string int8 int16 int32 type std_out

int64 uint8 uint8 uint32 uint64 float NULL double_float

std_in TRUE FALSE Tuple private this

Operators:

Mathematical Operators:

* + - / % ^ ++ --

Assignment:

= =+ =- =* =/ =^ =| =&

Compound Assignments:

The following compound Assignments are available within Learn.

=+ =- =* =/ =^

=% =| =& =>> =<<

Boolean Operators:

! || && == != >= <= > <

Bitwise Operators:

& | << >>

Other:

. * & [] : <type_name>() op

Types: Primitives

Primitives:

uint8	uint16	uint32	uint64	bool	char
int8	int16	int32	int64	float	double_float

Integer Types: All integer types explicitly state their size, and can be either signed or unsigned. With signed integers representing negatives numbers using two's complement.

Float Types: Learn has two floating point types. Floats represent single precision floating decimal numbers. While double_floats represent double precision floating point numbers.

Char: Learn adheres to the UTF-16 standards. This makes character types in Learn unicode compliant.

Bool: Booleans in Learn are represented using the bool keyword. Boolean types are 8 bits wide, and a string of zero bits represents false. While any other string is considered true.

Types: Non-Primitive

Non-Primitives:

String	Array	Slice	Tuple	Struct
Class	Interface	Map		

String: Learn uses limited dynamic length strings, consisting of a set adjacently stored char types. Being limited dynamic length means that strings in Learn can grow up to a finite length set during declaration.

<String_declaration> → string <identifier> [“[“ <int_type> “]”] [assignment]

Array: Arrays in Learn are fixed length and consist of one type. However Learn also provides the capacity for dynamic arrays through the use of pointers, and the key words new and delete. To allow for jagged arrays multidimensional array are implemented using arrays of arrays.

<array_declaration> → <type_name> <identifier> “[“ <int_type> “]” [<array_init>]
<arrray_init> → ‘{‘ ([<literal>][‘,’])* ‘}’

Types: Non-Primitive

Slices: serve as references to a subset of an Array.In memory a slice consists of a pointer, the length of the slice, and the total capacity. The pointer points to the first element of the array contained within the slice. The length gives the current length of the slice, while the capacity holds the maximum size of the slice.

EX:

```
//creates a subset of  a between the 3rd and 5th elements
uint32 a[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
Slice a_slice =  a[2:4]
```

Maps: Maps are Learn's associative array types. They consists of a <key,value> pair. Where keys can be any type that adheres to the hashable, and comparable interfaces.

EX:

```
Map a = {"first": 0, "second" : 1, "third" : 2. "4" : 4 }
```

Subprograms:

Functions:

Denoted using the function keyword. Then a comma delimited list of variables enclosed in parenthesis. They may not be nested, and must be defined in their own scope. They must return a type, with void being used to denote that a function that returns nothing.

<func_statement> → function <type_name> <identifier>(<arg_list>)
{ [statement]* return <type_name><end_line> }

Methods:

Methods in Learn behave in the same way as functions, except they are tied to an object. Their definitions are labeled with the method keyword. The keyword must then be followed by the name of the class it is to be tied to, and then its return type. The only exception to this rule comes from the constructor method, and the destructor method named destructor which has no return type.

<method_statement> → methods <class_name> <type_name>
<identifier>(<arg_list>)
{ [statement]* return <type_name><end_line> }

Conditional Statements:

If/Else:

In an if/else statement like other block statements the bracket characters (‘{‘ ‘}’) will be used to enclose the scope of each nested if/else block. This is done to help mitigate any possible grammar ambiguities. The else part of an if/else block is optional, in which case, should the conditional expression fail the then statement is jumped over, and execution continues as normal.

If (comparison) then{ statement} else { statement }

Switches:

Switch statements provide a cleaner way of writing embedded if/else statements. The expression is compared to each case statement, whose scope is delimited by the break statement. Which break out of the scope of the switch statement and returns control after the switch statement.

Ex:

```
switch(expression){
        case const1 :
                [statement]*
                 break;
                .
                .
         default{[statement]* break;}
```

Iterative Statements:

For Loops:

The only iterative statement in Learn is the for loop. It is capable of handling both conditional and iterative loops. A for loop must include parentheses enclosing its *initialization, condition, post* loop segments. For loops in Learn can omit both the *initialization*, and *post* segment allowing users to simulate a while loop, when needed. By not creating either segment we are left with a single statement over which to iterate, the *condition* segment. This *condition* can be either a logical or mathematical expression.

Ex:

for (*initialization*; *conditional*; *post*){
    [statement]*
}

Example: MyString Class

```
Class MyString (){
        private *char str
        private uint32 len
        private uint32 max
}

//constructor with default parameter assignment set to 0
method MyString(int N = 0){
        if(N == 0){
            this.str = null;
            this.len = 0;
                this.max=0;
          }esle{
              this.len = 0;
              this.max = N;
              this.str = new char[len];
          }
        Return void;
}
```

Example: MyString Class

```
//destructor called whenever an object goes out of scope or at the end of program //execution
method Mystring destructor(){
        Str = delete char[this.max]
        this.len = 0; this.max = 0;
}

  //return char at index or null
  method Mystring char at(uint32 index){
          if(index >=0 && index < this.max){
                  return *this[index];
          }
          return null;
  }
```

Example: MyString Class

```
//append a char to the end of object
// calles resize if string is already at max size
method MyString void append (char rune){
        if(this.str == null){
                this.str = new char[1];
                this.Str[0] = rune;
        }esle {
                if( this.len == this.max){
                        this.resize(this.max);
                         this.append(rune);
                }else{
                        this.str[this.len] = rune;
                         This.len++;
                  }
               }
        Return void;
}
```

Example: MyString Class

```
//resize string while maintaining original string.
Method MyString void resize(uint32 size){
        Mystring tmp = *this;           // making copy of object
        uint32 N = this.max + size;
        Uint32 n = this.len;

        *this = new MyString(N);
        for(int i = 0; i < n;i++){
                *this[i] = temp[i];
        }
        return void;
}
```

Example: MyString Class

```
//method to split a string into two substrings. Searching for the first demilnator starting from //index i.
// input: char delimiter default=  “ “
//          uint32 i default = 0
//          uint32 j default = this.max
method Mystring Tuple[ s1:MyString, s2:MyString] split (char delim = “ ”,uint32 i = 0,uint32 j = this.max)
{
        Uint64 start = i
        Uint64 end = j

        for(; i < j || this.str[i] != delim; i++){}

        Mystring tmp1 = this.substring(start,i)
        Mystring tmp2 = this.substring(i,j)

        Return Tuple[s1: MyString = tmp1, s2: Mystring = tmp2]
}
```

Example: MyString Class

```
//operator overloading to return  a string consisting of x concatenated with y.
 op Mystring  Mystring + (MyString x, MyString y){
          Return Mystring val =  x.append(y);
}

Op Mystring char [](Mystring x, uint32 index){
                    Return x.at(index);
}
```