Learn

Introduction:

The main goal of this language is to provide new programmers with a simple enough tool set that they don't feel overwhelmed by the amount of syntacial rules and primitives. In keeping with this philosophy we will limit our use of reserved words as much as possible. While still providing tools to allow programmers to create their own types. And by defining our keywords and reserved words as clearly as possible. Learn has all base mathematical, boolean, and bitwise operators. The language will be explicitly typed meaning no variable can be used before it is declared and its declaration must begin with a type name. The language will also be strictly typed, to prevent unwanted user errors, in the missassignments of types and loss of data that can follow coercion and conversion between types. Basic mathematical operations are all included. With the addition of ^ used to denote exponential values. This is mainly added to boost the expressivity of the language.

Problem Domain and Approach:

Learn is a language whose main purpose is its name. As creative or not as that might seem, the directness and simplicity of this naming is at the core of the language philosophy. It is meant to help new programs learn the core concepts of computer programming. Providing a core set of features and data types. Learn is a general purpose high level language, inspired by C, and Java. It is an imperative, object oriented language. Learn is an educational programming language that focuses on a learn by doing approach. In keeping with this ideal, simplicity, orthogonality, writability, readability, and expressivity will be of the utmost concern. Simplicity will be accomplished by providing a smaller set of types, objects and looping mechanisms. With

the goal of not overwhelming users with options and a complex web of interrelated types and structures. While still providing the core fundamentals needed in any language, and enough orthogonality to allow the programmer to handle a large set of problems. It's my belief that one of the main obstacles for many new programmers is a sense of being overwhelmed by the naming conventions and tool sets provided by most programming languages. It's this problem that Learn hopes to address. Therefore by necessity we will focus on writability, and readability as much as possible. This focus on simplicity, orthogonality, readability and writability should lend itself to the overall expressivity of the language. This is also true of the cost of Learn, by making the language simple and easy to read and write the cost of implementation, and training should go down.

Specification:

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:

* + - / % ^ ++ --

Boolean Operators:

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

Bitwise Operators:

& | << >>

Assignment:

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

Other:

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

Ex:

```
int16 x = 2 ^2          // x = two raised to the second power.
```

Bitwise operators will be denoted using the single characters reserved words; & will denote bitwise AND and | will denote bitwise OR.

Ex:

```
Bool and = a & b        // & represent bitwise and
```

The only exception to this rule is bitwise shift either left or right, these will be denoted with double less than, and double greater than, respectively.

Ex:

```
int16 x = 2 << 1          // left shift  2 by 1, same as 2*2
```

The language will be Unicode compliant, so that both character type primitives and string types support unicode encoding. This is mainly to provide additional compatibility for the language, by allowing the user to print and parse most human languages. The std_out keyword will be used as an identifier for the std_out() function. This function will handle all standard output. And the std_in keyword will be used as the identifier for std_in() which will handle all standard input as well as type conversion for the required type.

Lexical Rules:

All programs begin with a statement, Statements will consist of variable declarations, or block statements which can be nested. Both the characters semicolon ";" and newline ("/n") denote the end of statements, however multiple statements can not be combined onto a single line without a semicolon to separate them.

<statement> → <var_declaration><end_line> | <block_statements><end_line>

<end_line> → ";" | "/n"

Ex:

```
uint8 y = 9
uint8 i = 0        //note  that no semicolon is used here or above.
```

Ex:

```
uint8 a = 7; uint8 n = 9;
```

Comments are denoted with '//' double backslash, every character after the second backslash will be considered a comment up until a new line character is found.

<comment> → //{a-z}{0:9}<end_line>

Ex:

```
// this is a comment
```

All variables must begin with a letter, underscores are allowed but never in the beginning of a variable name. This is to allow both camel case and snake case variable names. Variables are all explicitly typed meaning a type name must begin any variable declaration followed by an identifier with an optional assignment. User defined identifiers for variables, funcs, interfaces etc must not include special characters. All typing will be done statically, this means that all variable types are known during compile time and that a variable's type is fixed during run-time. Type casting is allowed, but any type casted variable will be treated as a RHS value. This is done to keep execution more efficient and also to help prevent runtime errors tied to miss matched types. Static typing also gives the language improved speed, clearer readability and more dependability. All of which is a benefit to new coders. All variables must be declared before use, even global variables and constant global variables. This is to avoid undefined reference errors and to reinforce strong typing. Note that within the same scope no two identifiers can be the same. The newest declaration will overwrite the older.

<var_declaration> → <type_name> <identifier> [<assignment>] |<pointer_declaration>
| <array_declaration> | <map_declaration> | <slice_declaration>
|<tuple_declaration>

<assignment> → = <expression> | = <literal> | =<identifier> | =<op>

<expression> → <identifier> <op> <identifier> | <expression><op><expression>

| <unary_op><literal> | <unary_op><identifier>

|<identifier> | <literal>

<op> → <binary_op> | <bitwise_op> |<unary_op>

<binary_op> → * | + | % | ^ | <boolean_op> |<bitwise_op>

<boolean_op> → && | || | == | != | > | < | >= | <=

<bitwise_op> → >> | << | & | |

<unary_op> → ! | - | ++ | --

<identifier> → a-z{a-z}{ _} {a-z} | {unicode char} | <global_identifier>

| <global_const_identifier>

<literal> → <lit_num> | <lit_bool> | <hex_literal> | <lit_char>

<lit_num> → 1-9{0-9}[.]{0-9}

<hex_literal> → 0x{[0:9][A-F]}

<lit_bool> → | FALSE | TRUE | NULL

<lit_char> → ‘ any unicode character ’

<type_name> → <int_type> | <float_type>| bool |char

| <identifier> | <derived_type>

<int_type> → uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

<float_type> → float | double_float

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

<buffer_type> → map | array | slice

Ex:

uint8 unsigned_int = 34

Ex :

int8 a

Ex:

```
bool test = false;
```

Ex:

```
type foo int;
foo x = i;
```

Variable names beginning with a capital are global in scope no matter where they are declared.

<global> → <type_name><global_identifier>

<global_identifier> → A -Z[_]{a-z}

Any variables consisting of all capital letters and no underscore will be both global and constant. Meaning that the variable's value may never be changed once declared, any global variable's scope will extend to the end of the program. We use syntax to distinguish between scopes in order to limit the use of const. As well as to help readability by making it fast and easy for a user to identify between constant globals and global variables outside of their declaration.

<global_const> → <type_name> <global_const_identifier>

<global_const_identifier> → A -Z{A-Z}

Ex:

```
float PI = 3.1415926
x = 0
if( x == PI){
 std_out(x + " = " + PI)
}
```

Storage:

Storage of local and global variables will be statically allocated. This is for both speed and simplicity. All variables declared within a scope will be implemented as stack-dynamic variables

as in java and C++. Meaning their types will remain statically bound but will only be allocated during the execution of a block. This is to allow the use of recursive functions and methods. All pointer types will be dynamically allocated onto the heap at compile time. Making them explicit heap-dynamic. This is to allow the use of dynamic arrays and data structures like linked lists. In the hopes that as programmers get a feel for programming, they can explore more complex data structures, while also gaining a concrete understanding of memory use if desired. Users can handle their own garbage collection using the new and delete reserved words. This will be further explained in the section dealing with pointers. By default for users wishing to not deal with their own garbage collection we will use reference counters. Reference counters will count all the pointers currently pointing to a specific address in memory. Everytime a pointer is allocated for a specific address the appropriate counter will be updated. Once a pointer goes out of scope or is no longer reachable the counter will be decreased. If the counter reaches zero the memory address is deallocated. The use of both garbage collection and user dependent memory allocation and deallocation is again to provide flexibility in both use and runtime speeds. While the garbage collection may slow down execution, this is still an educational programming language, and forcing new users into the deep end can scare off newer students from pointers. Therefore garbage collection is implemented as a back up for those new users.

Scoping:

Blocks will begin with an open bracket character "{" and will end with a closed bracket "}". The biggest benefit of this approach is in readability, this is of most importance when it comes to beginner programmers. By allowing users to avoid frustrating indentation errors. Functions can

not be nested. This is done to keep readability, in a nested set of functions the scope of variables can be more difficult to detect then in non-nested functions, that require the passing of variables, or the explicit declaration of each variable within its block. Block statements will consist of boolean statements, switch statements, loop statements (of which there is only a for loop), and function statements. And each type of block statement has their own local scope. Variables within each block will be static-dynamic as mentioned earlier. Blocks other than function statements can be nested inside larger blocks similar to C, and java. Note that in these cases the scope of a variable will be determined by its location. Variables in a larger block scope will be visible to the inner scope. For example the in an embedded for loop, any variables declared in the outer loop are visible in the inner scope, but not vice-versa. Variables needed in a function must be either passed to the function through the calling sequence, or must be declared within the scope of the function (this includes global variables as they are visible in all scopes). Note that any function statement may have an optional type name preceding the func keyword. Any function not including a type name will be considered void, meaning it returns nothing. Any function with a type name must contain a return statement of the same type.

Blocks:

<statement> → [<var_declaration>]*[<block_statment>]*

<block_statement> → <boolean_statement> | <loop_statement>

| <switch_statement> | <struct_statement>

<struc_statement> → struct <identifier> “{“ <statement> “}”

<boolean_statement> → if (<comparison>) “{“ <statement> “}” [else “{“ <statement> “}”]

<switch_statementt> → switch(<expression>) “{

{case(<literal>) “{“ [<statement>] break<endline> “}” }

[default “{”<statement> “}”] “}”

<loop_statement> → for(<var_declaration>;<comparison>;<expression>) “{“ <statement>“}”

| for(<;<comparison>;) “{“ <statement>“}”

<func_statement> → function <type_name> <identifier>(<arg_list>)

{ [statement]* return <type_name><end_line> }

<comparison> → <expression> <boolean_op> <expression> | <boolean_val>

Ex:

```
for(int8 i = 0; i < n;i = i +1){
        //code
}
```

Ex:

```
if (true){
        //code
}Else{
        //code
}
```

Primitives:

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

Integer types:

All integer types will be labeled by their bit size, this is meant to not only help prevent conversion and overflow errors, but also to help reinforce an understanding of the underlying memory involved. These sizes range from 8bits to 64 bits. Both signed and unsigned integer values are available. With a "u" used to label an integer type as unsigned. Here the goal is to create type names that are as explicit as possible and while not being cumbersome to use. All integer types both signed and unsigned will be implemented by a string of contiguous bits in memory, addressed based on the leftmost bits (LMB) location. Signed values will use LMB to indicate sign, with a 1 meaning negative and a zero denoting positives. All negative numbers will be represented using 2's complement. The means that signed integer value range will be $(-2^{n-1}, 2^{n-1} - 1),\ n = 8, 16, 32, 64$. While the value range for unsigned integers will be $(0, 2^{n} - 1)$, n = 8,16,32,64

Ex:

```
int8 x = 4;
```

0	0	0	0	0	1	0	0

The image above shows how all 8int types will be represented in memory. In this case it shows number 4 as it would be stored in memory. The MSB location will be used as the memory address of the variable itself. All other number types will be represented the same way, except wil more bits based on the integer type used.

Floating Point Types:

Floats will be of two types, denoted float or double_float, representing single and double precision float values. In keeping with IEEE standards single precision floats will be represented in 32 bits, and double precision floats will be represented in 64.

Ex:

```
float x = 2.004
```

The follow is a single precision floating point number as it would be represented in bits.

Sign bit	Exponent	Fraction
1 bits	8 bits	23 bits

Ex:

```
double_float  x = 4.10
```

Below is a representation of a double precision floating point number.

Sign bit	Exponent	Fraction
1 bits	11 bits	52 bits

Character Types:

All characters will be represented using UTF-16, this is to allow for unicode compliance. This means that characters will be stored in a minimum of 2 bytes and a max of 4 depending on the character being represented. We want to use this scheme in order to be as efficient in our memory use as possible. As well as giving us all the advantages of UTF-16, namely being able to parse languages left to right, or right to left, as well as the normalization of words that UTF-16 uses will be beneficial to parsing different human languages.

Ex:

_____ char x = ‘x’

Boolean Types:

Booleans will be represented using the smallest possible addressable memory size. This is one byte or four bits. There are two true boolean types TRUE and FALSE Any value other than 0 represented will be true, and only exactly 0 will be considered false. The only other value to represent false will be the special type null to be discussed later on.

Ex:

bool this_is_a_bool = TRUE;_

Null:

The special primitive type NULL will be used to identify empty pointers and object types. This will be represented in one word of eight bits and will be denoted by a set of all zeros just like the false boolean type. The main purpose of null is to allow us to check for null pointers, and objects like structs. Null will be represented in memory the same as a false boolean.

Pointers:

<pointer_declaration> → <type_name> <identifier>‘*’ [<pointer_allocation>]

<pointer_allocation> → “new” <type_name> [<identifier>] [“[“<int_type>”]”]

<pointer_deallocation> → “delete” <identifier> “[]”

Pointers will be used to allocate dynamic memory and can be assigned to any primitive or derived type. They will be declared with a * character directly preceding the base type name. As stated previously they will be allocated and deallocated dynamically allocated onto the heap.

They will be stored in memory as a byte long string of bits whose value represents the address of the variable/object being pointed to.

EX:

```
*uint8 ptr;
```

Reference:

<reference_declaration> → <type_name> '&' <identifier><assignment>

Reference types will be used as a type of aliasing to objects and functions. They will be denoted by their type followed by an ampersand &. For references to functions the return type of the function must be the same as the referencing variable. All variables of the object can be accessed using the dot (.) operator. This will provide access to the public variables declared in an object. For function reference types, the parameters of the function must be passed into the reference using parenthesis. Unlike pointers a reference declaration must include an assignment and the assignment may never be null. This implies that a reference can only be made to an already existing object or variable.

EX:

```
struct employee {
   int64 id
  float  salary
}

Func main(){
        Employee joe
        Employee& r = joe

        joe.id = 23049
        std_out(joe.id)
        r.id = 10977
        std_out(joe.id)
```

}

Non-Primitives:

String Array Slice Tuple Struct

Class Interface Map

Strings:

Strings will be implemented as limited dynamic length strings. A string can grow and shrink up to its initial fixed length. They will be stored in adjacent memory cells as arrays of characters, with the max length of the string stored as well. Strings will have built in conversion functions for each primitive type. As well as base string operations, i.e copy, concatenation, substring matching, etc. If a user wishes to convert their own objects to strings, then the object must conform to the printable interface. Which works by providing the print() function and the toString() function. The first of which provided a print method and the second provides a conversion function.

<String_declaration> → string <identifier> ["[" <int_type> "]"] [assignment]

Arrays:

<array_declaration> → <type_name> <identifier> "[" <int_type> "]" [<array_init>]

<arrray_init> → "{" ([<literal>][","])* "}"

Arrays can consist of any referenceable type and will be of fixed length. Making all base arrays static arrays. This means that an array's length is fixed for the entire program once it has been initialized. Arrays can be initialized during allocation as shown in the example below. If an array is initialized then the size of the array can be implicitly found. For example the following array declaration creates an array of size five, which is found by counting each of the items filling the

array. Note though that an array's type must always be stated explicitly during declaration, and any value initialized to any of its elements must be of that same type.

EX:

```
int8 a[] = {0,1,2,3,4} //creates an array of 8 bit ints of size 5
```

However the array size can still be stated explicitly in the initialization. But the elements initialized must be no more than N elements. And any values left uninitialized will be set to their type's default value.

EX:

```
/*creates an array of 8 bit ints of size 32, where
only the  first 5 elements are initialized.*/
int8 a[32] = {0,1,2,3,4}
```

Dynamic arrays are possible through the use of pointers. (For dynamic arrays see the section on pointer types, and for information of garbage collection see the storage section .). All of the elements of an array must be consistent, that is to say they must all be made up of the same type. (For collections of different objects see either the tuple section). Multidimensional arrays will be implemented as arrays of arrays, allowing for jagged arrays. The bracket operator [], will be used to reference a specific index within a given array. For multidimensional arrays multiple brackets are used to access each level of the array. Subsets can be extracted from an array using the colon operator (:) to denote range. This type of operation produces a slice.

EX:

```
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]     //creates a subset of a between the 3rd and 5th elements
```

Note that if either the first or last value in the colon delimited range are missing then they are implied. If the first is missing i.e [:10], returns a slice consisting of index's 0→10, and [0:]

returns a slice made up of every element in the array beginning at index 0 and ending at the last element of the original array. Slices can not exceed the range of its original array.

Array operations:

Len() :

Returns the length of an array, when used on a multidimensional array returns the length of the array currently being referenced.

Subset(32uint i, 32uint max):

Returns a subset of the array, creating a slice from i to max, where max by default is the full lenght of the original array. The Bracket operator in combination with colon operator has the same effect. As illustrated in the previous example.

Equals(array a):

Returns a boolean value resulting from the comparison of two arrays, the calling array and the array passed as an argument. Arrays are only equal if every one of their elements are exactly the same, and they are of the same size.

Sort():

Sorts the calling array in ascending order. Comparison of elements will be based on the type of array. The object used within an Array must adhere to the comparable interface if you wish to sort the array, otherwise an exception will be thrown.

Reverse():

Reverses all the elements in the array.

Find(<type> e):

Returns the index value of the first found element within the array. Note the type of element (e) and the calling array must be the same. Returns -1 if not found.

FindLast(<type> e):

Returns the index value of the last found element that matches e within the array. Returns a -1 if not found. Note the type of element and the calling array must be the same.

Slice:

Slices will consist of subsets of larger arrays. As such they must be initialized to some range of an array. The type of the slice is inherited from the original array. In memory a slice will consist of a single 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.

Tuples:

<tuple_declaration> → “tuple” <identifier> <tuple_init>

<tuple_init> → “{“ ([<char_lit> *“:”] <type_name> “,”)* “}”

Tuples are arrays of elements stored in adjacent memory cells that do not consist of the same type. They can be used to define a data type, in this they resemble java records. Their main purpose will be to pass and return multiple values into a function or method, or to manipulate collections of data that represent some larger object. Elements can be accessed numerically using the bracket operator ([]), filled with a valid index corresponding sequentially to each tuple

element. Or they can be named using the colon (:) operator during instantiation, then referenced using the dot operator (.) followed by the reference name. This allows tuples to be used as records, where each element of a tuple can be a different object that can be referenced and manipulated.

EX:

```
Tuple Name = {first : string, middle : string , last : string};
Tuple Employee = { Name, Salary : uint32, ID: uint32};
Employee Joe;    /*defines an employee named Joe, with access to all the fields defined
                   in Employee*/
//prints Joe's Name,Salary, and ID on three separate lines.
std_out(Joe.Name);
std_out(Joe.Salary);
std_out(joe.ID)
```

Maps:

<map_declaration> → "map" <identifier> [<map_init>]

<map_init> → "{" (<literal> ":"<literal>","")* "}"

| "{" (<identifier> ":"<literal>","")* "}"

| "{" (<literal> ":"<identifier>","")* "}"

| "{" (<identifier> ":"<identifier>","")* "}"

Maps are arrays, whose elements consist of hashed key, value pairs. The hash function used will depend on the type of key being used. This also means that user defined types can be used as keys. To do this however the user defined type must conform to the Hashable interface. This means the type must have a definition for the hash(), function. To reference any value within a map the bracket operators are used, much like with an array, except the key need not be an

integer type. Like an array, maps can be initialized during assignment. However unlike arrays, maps are of dynamic length, making it quick and easy to add elements. In practice each map is dynamically allocated and deallocated using garbage collected when needed. It's important to note that though the keys and values need not be of the same type, all the keys must be of the same type, and so must all the values.

EX:

```
Map a = {“first”: 0, “second” : 1, “third” : 2. “4” : 4 }
std_out(a[“first”]);                //prints the first element of the map to the console.
```

Mathematical Expressions:

All mathematical operators carry traditional precedence values, and have left associativity, except for both the exponential operator (^) ,and any unary operators (i.e ++,--,-,!) which have right associativity. Note however that parenthesis can be used to set precedence within an expression. All expressions must be written in Infix notation, as this is likely what new programmers are accustomed to using. Functions are allowed to have side effects. In order to try and minimize errors however any functional calls within an expression take top precedence, and are therefore evaluated first. As mentioned in earlier sections, to preserve strict typing, expressions are not allowed to have mismatched types. However explicit type casting can be done in which case the variable is either widened or narrowed depending on the conversion taking place. Even with explicit type casting however all the final types involved in the expression must be the same. The user should prepare for and be aware of the loss of information due to any narrowing that may occur when converting from a larger data type to a smaller one.

Below is an example of type casting in Learn. The radius variable is casted to a double_float widening the variable and allowing the expression to be valid. It's important to note that type of casting is limited to primitive types only.

EX:

```
Doube_float pi = 3.1415926535897932
int64 radius = 5
Double_float area =  pi * double_float(radius) ^2
```

Logical Expressions:

The precedence of the boolean operators is as follows, not (!) has the highest, while and (&&) and or (||) have the same level of precedence. The latter two have left associativity while the not operator has right associativity. With the same precedence rules being followed by their bitwise counterparts. Like mathematical expressions parenthesis can be used to overwrite traditional precedence rules. In both logical and mathematical expressions short-circuiting is allowed. For example the following boolean expression ends its evaluation at the first or expression if it is correct. If the first statement a ==c is true then the entire expression is true, therefore there is no need to evaluate the rest of the boolean expression.

EX:

```
//...code
if( a == c ||  (!a &&b) && c){
//..code
}
```

The same short circuiting applies to mathematical expressions, whenever possible if a zero is found that propagates through the entire expression. Rather than completing the evaluation the expression ends evaluation and returns a zero.

Overflow and UnderFlow:

Whenever an overflow occurs because the result of an expression is too large to be stored in the original type then the resulting expression will return the max value associated with its type. Whether it be an integer type or float types. In the case of underflow, the resulting expression will return the smallest possible negative number provided by the data type. In the case of all unsigned integers an underflow will produce a zero. Note that whenever a division by zero is encountered however then an automatic exception will be thrown.

Overloading:

Operator overloading is allowed, for example the + operator can be used in a typical mathematical expression to represent addition. But it is also used to represent concatenation of strings. Furthermore users are allowed to define operators for their own data types. Though allowing users to define their own operator behavior can make code harder to read, its advantages are worth allowing. Though it is important to note that users are not allowed to overload operators that deal exclusively with any primitive or language provided data types.

EX:

```
class vector{
//..class code
//…
//…
Vector op +(Vector x,Vector y); //operator overloading for inner product of two vectors
}
```

Assignments:

Neither simple assignments nor multiple assignments are allowed within the Learn language. In order to avoid multiple assignments as a side effect assignments can not be used as expressions.

Conditional Assignments:

Conditional assignments are allowed within Learn, the following example illustrates their syntax.

EX:

```
//...code
a = (c ?= d) 25 : 0;
```

Where the assignment states:

if c == d then a = 25 or a= 0.

Compound Assignments:

The following compound Assignments are available within Learn.

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

=>> =<<

Each an abbreviation of an assignment of the following type.

EX:

```
int8 a = 0;
int8 b = 5;
a =+ b;          //assign  a + b to a
std_out(a);      //print a
```

Where the operator following the equals sign (=) denotes the operation being used.

Unary Assignments:

Learn provides the following unary assignments ++, and --. Like C both can be either prefix or postfix operators. With the first representing increment and the second representing decrement.

EX:

```
uint64 x = 5
uint8 n=0

std_int(n)
for(uint8 i = 0; i <=n;x++)
        std_out(x)
```

In the above example the x++ is equivalent to x = x + 1.

Control Flow:

Control structures in Learn are composed of if/else statements which can be nested, or switch statements which can not. Nested switches are excluded in order to avoid unreadable nested switches.

if/else statements:

For the syntax and grammar rules associated with if/else (boolean) statements see the blocks section of the report. 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. Below is an example if/else statement.

Ex:

```
if(i == 10){}
else{}
```

The general semantics of Learns if/else statements can be summarized as follows:

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

Meaning if the comparison is true, then the statement enclosed by the if part of the statement is executed, otherwise the statement enclosed by the else statement is executed. However 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.

Switch Statements:

Learn switch statements will consist of the switch keyword followed by an expression surrounded by parenthesis. This expression is then compared directly to each case statement, which must consist of some constant type. The total scope of a switch statement is denoted by using the bracket characters like if/else statements. Each case statement can consist of any number of expressions, assignments, or block statements. Every case statement must end with a break statement, which breaks out of the scope of the switch statement. Therefore the execution flow of any switch statement is restricted to just one selectable case. For every switch statement a default selectable segment must be present. The default statement must also end in a break statement. The default statement is used to catch any values not specified using case statements, and to reinforce robust code.

Ex:

```
switch(expression){
        case const1 :
                [statement]*
        break;
        case const2 :
                [statement]*
        break;
        case const1 :
                [statement]*
        break;
        //...
        //...
        //...
      case const1 :
                [statement]*
        break;
        default:
```

```
                [statement]*
        break;
}
```

Iterative Statements:

Learn borrows its litterative statement style from both C, and Go. That is to say Learn only has one type of loop statement; the for loop. This for loop is capable of handling both conditional and iterative loops. A for loop must include parentheses enclosing its *initialization, condition, post* loop segments, described below.

For Statements:

Like C, 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, including subroutine calls that return primitive values. This allows the for loop to take on the behavior of a conditional loop. Omitting the *initialization* section alone is also allowed for variables needed outside the scope of the loop. The *initialization* segment consists of a single variable declaration when not omitted, whose scope is local to loop. Nesting for loops is allowed and variables created in an outer loop are visible to any subsequently nested loops, but not vice-versa. The *post* section consists of any expression. Like C, Learn has no explicit loop parameters, instead the flow of execution is as follows: the *initialization* segment is evaluated once before execution, while the *conditional* is evaluated at the top of every loop, and the *post* is executed at the end of each loop. The general form of a for loop is as follows:

Ex:

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

Subprograms:

Learn is composed of functions and methods. A function is any standalone subprogram, which accepts a comma delimited list of arguments. The signature of a function must declare a variable's type followed by a unique identifier, with return types explicitly declared. If the function is not meant to return a value then it must be labeled void. Any non void function must end in a return statement. Local variables are stack dynamic, meaning they are allocated during the execution of a function and unbound after. Stack dynamic variables improves execution time, and allows for recursive function calls. Because local variables can not escape their scope any returned value is tried as a RHS value.Values can be either passed by value, or by reference. In the case of a pass by reference, variables must be declared reference types in the subprograms signature using the & operator. Subprograms may not be nested, meaning they must be defined within their own scopes. Though within its body, a function can call any number of functions. As mentioned in the mathematical expression section, functions are allowed to have side effects. A function may return only one type of variable, but we can return object types like Tuples. Effectively allowing programmers to simulate returning multiple values. Overloading of functions is allowed, but the signature of each function must be unique. Unique in either its list of arguments or return type. Operator overloading is implemented through the definition of functions declared with the op keyword. Because operators act like functions they do not have direct access to an objects private variables. Instead all access and modification must be done

through the use of mutators and accessors. Methods are functions which are tied to and called by an object. For more on both methods and functions see their respective sections below.

Functions:

Function definitions are labeled using the function keyword. Followed by the name of the function, then a list of the function's arguments enclosed in parenthesis. A function's body must be enclosed in brackets. Learn uses a combination of positional and keyword parameters, and you may set an arguments default value in its signature. When keywords are used to pass a value into a function during execution, any subsequent variables must also be keyword assigned. When defining a function an argument is assigned any value; any following arguments must also set default values.

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.

Appendix:

Below is part of an example of a user defined string class called MyString. Unlike built-in Strings the MyString class is able to grow and shrink dynamically. It showcases Learn classes, loops if/else statements, dynamic allocation, method/function declarations and implementation, Tuples, and recursion.

Ex:

```
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];
        }
}

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

//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++;
                }
              }
}

//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];
        }
}
```

```
//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]
}

//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);
}
```