A Language with Beautiful Syntax
(ALBS)

Language Reference Manual

Programming Languages and Translators Spring 2016
March 5, 2016

Name	UNI	Role
Brennan Wallace	bgw2119	Language Guru
Suhani Singhal	ss4925	Product Manager

Table of Contents

1. Introduction

This manual describes the ALBS language - a functional language with immutable variables and C-like syntax, that is compiled down to LLVM.

`Throughout this manual, sample code is written in monospace font, as this is.`

2. Lexical Conventions

2.1. Tokens

There are four types of tokens: keywords, identifiers, constants, operators, and punctuation. Blanks, horizontal and vertical tabs, newlines, formfeeds and comments as described below (collectively know as white space) are ignored except as they separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

2.2. Comments

Comments are single line and the line must start with two '/' characters with no space between them though any amount of space can be before them as well as code that is evaluated normally. Anything inside a comment is ignored for the purposing of compilation and create if the abstract syntax tree.

To Illustrate:

```
return 0; //this is a comment //this too
this is not a comment
/ neither is this
```

2.3. Identifiers

Identifiers are sequences of alphanumeric characters and underscores that are largely used as variable names. A regular expression that can be used to define them is `['a'-'Z'|'0'-'9'|'_']+`.

2.4. Keywords

ALBS has a set of reserved keywords and function names that cannot be used as identifiers for variables. This is a list of all reserved keywords:

Data Types	Control Flow	I/O functions	List functions	Other
bln	rtn	print	head	true
flt	if	input	tail	false
int	else		rightReduce	
chr	main		leftReduce	
void			listLength	
			listGet	
			listAdd	

2.5. Constants

2.5.1. Integer literal

Integer literals are a sequence of numerals without decimal point. Not having a decimal point is crucial as this distinguishes them from doubles. A hyphen may precede the numerals to indicate that the value is negative. Note: -0 is will be converted to 0 during parsing. A regular expression for ints can be defined as follows:

```
['0'-'9']+
```

An integer literal made of multiple zeros without a non-zero numeral is converted to a single 0 during parsing. An integer literal with zeros before (to the left) of the first non-zero is equivalent to an integer literal of the same sequence without the preceding 0's.

To illustrate:

```
int zero = 0000;//zero is 0
int positive_one = 1;//positive_one is 1
int negative_one = -00001;//negative_one is -1
```

2.5.2. Float literal

Unlike integers floats can and are required to have a single decimal point somewhere in the sequence of digits but are otherwise similar. A regular expression for floats can be defined as follows:

```
-?([‘0’-’9’]+’.’[‘0’-’9’]*|[‘0’-’9’]*’.’[‘0’-’9’]+)
```

If one side of the decimal has no characters it is equivalent to a single 0 being there. An float literal made of multiple zeros without a non-zero numeral is converted to a single 0 during parsing. An float literal with zeros before (to the left) of the first non-zero is equivalent to a float literal of the same sequence without the preceding 0’s.

To illustrate:

```
int zero = 0000;//zero is 0
int positive_one_point_one = 1.100;//equivalent to 1.1
int negative_one_point_one = -00001.10;//equivalent to -1.1
```

2.5.3. Character literal

Character literals use a single quote before and after the character. Characters generally represent themselves (‘a’ is the literal for the character representing the letter a). However, the following table shows how some characters are defined using a backslash and another character. This table largely pulled from The C Programming Language 2nd Edition as we feel it is important to support most of the same “special” characters in the same way.

Character	**Literal w/Quotes**
newline	‘\n’
horizontal tab	‘\t’
vertical tab	‘\v’
backspace	‘\b’
carriage return	‘\r’
backslash	‘\\’
single quote	‘\’’
double quote	‘\”

To Illustrate:

```
chr a = 'a';
chr newline = '\n';
//both of these set the variables
//to their namesake values

rtn 'c';
//shows a non assignment usage
```

2.5.4. Boolean literal

Boolean literals come in the form of two reserved keywords “true” and “false” (without the quotes).

To Illustrate:

```
rtn true == true; //returns true
rtn true != false; //returns true
rtn false == false; //returns true
rtn true; //returns true
```

2.6. String literal

String literals are declared using a sequence of characters (without single quotes) that start and end with a double quote. Such literals are converted into lists of characters where the order of the list reflects the string literal. As such , string literals have the type of <**char**> which is a list of characters.

To Illustrate:

```
<char> ex1 = "example";
<char>  ex2 = ['e' 'x', 'a', 'm', 'p', 'l', 'e'];
ex1 == ex2; // returns true
```

3. Expressions

3.1. Infix Expressions

Operators in expressions are evaluated in according to the follow precedence table where a lower number is evaluated before a higher number and on a tie are evaluated as according to their associativity.

Precedence	Operator (Symbols)	Operator (Names)	Associativity

0	({expression})	Parentheses	Left to Right
1	({type name}) - !	Cast Negation Not	Left to Right Right to Left Right to Left
2	* / %	Multiplication Modulo Remainder	Left to Right Left to Right Left to Right
3	+ -	Addition Multiplication	Left to Right Left to Right
4	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to Right Left to Right Left to Right Left to Right
5	== !=	Equal Not Equal	Left to Right Left to Right
6	&	And	Left to right
7	|	Or	Left to right
8	=	Assign	Right to Left

3.2. Operators

Operator tokens indicate operations perform an action that takes the value of one or two operands (a literal or variable) and returns a new value. There are two category types number of operators (unary or binary) and output types.

The following tables describe the operators. Capital letters represent appropriate variables or literals. Descriptions in curly brackets should not be taken literally.

3.2.1. Unary Operators

Name	Notation	Description
Negation	-A	Can be applied to a float or an integer and returns a new float or integer that as a value equal to the original multiplied by -1.
Not	!A	Can be applied to a boolean and returns the opposite value of that boolean.

Cast	({Type to Convert To}) A	Takes a value and returns a new value of a different type as according to the following table. Note that if a cast is not listed in the table it is not permitted and attempting to do so will return an error.

3.2.2. Conversion Chart

Input Type	Output Type	Description
int	float	Converts an integer to a float.
float	int	Converts a float to an integer. Fractional amounts are truncated.
char	int	Characters are converted to ints that are equal to their ASCII codes.
int	char	Ints are converted to characters that have a ASCII code equal to that int. If there is no such ASCII character an exception is thrown.

You can also convert lists from one type to another.

Input Type	Output Type	Description
<int>	<float>	Converts an integer list to a float list.
<float>	<int>	Converts a float list to an integer list. Fractional amounts are truncated.
<char>	<int>	Character list is converted to integer list that are equivalent in their ASCII codes.
<int>	<char>	Integer lists are converted to character lists having the ASCII code equal to that int. If there is no such ASCII character an exception is thrown.

3.2.3. Arithmetic Binary Operator

Name	Notation	Description
Addition	A+B	Can be applied to two floats or two integers and returns a new float or integer (output type is the

		same as the input types) that has a value approximately equal to mathematical sum of the two inputs. Output if overflow occurs is not guaranteed.
Subtraction	A-B	Can be applied to two floats or two integers and returns a new float or integer (output type is the same as the input types) that has a value approximately equal to mathematical difference of the two inputs. Output if overflow occurs is not guaranteed.
Multiplication	A*B	Can be applied to two floats or two integers and returns a new float or integer (output type is the same as the input types) that has a value approximately equal to mathematical product of the two inputs. Output if overflow occurs is not guaranteed.
Division	A/B	Can be applied to two floats or two integers and returns a new float or integer (output type is the same as the input types) that has a value approximately equal to mathematical quotient of dividing the two inputs (A divided by B). The value is truncated. Output if overflow occurs is not guaranteed.
Modulo	A%B	Can be applied to two floats or two integers and returns a new float or integer (output type is the same as the input types) that has a value approximately equal to mathematical remainder of dividing the two inputs (A divided by B). Output if overflow occurs is not guaranteed.

3.2.4. Logical Binary Operators

Name	Notation	Description
And	A & B	Takes two booleans and returns true if both operands are true.
Or	A | B	Takes two booleans and returns true if at least one operand is true.

3.2.5. Comparison Operators

Name	Notation	Description
Equals	A == B	Takes two operands of the same type and returns true if they are identical in value and false otherwise.
Not Equals	A != B	Takes two operands of the same type and returns false if they are identical in value and true otherwise.
Less than	A < B	Takes two operands of the same type. Returns true if they are floats and integers and A is mathematically less than B. Returns true if they are characters and A precedes B in the ASCII character table. Otherwise false is returned.
.Greater than	A > B	Takes two operands of the same type. Returns true if they are floats and integers and A is mathematically greater than B. Returns true if they are characters and A follows B in the ASCII character table. Otherwise returns false.
Less than or equal to	A <= B	Takes two operands of the same type. Returns false if they are floats and integers and A is mathematically greater than B. Returns false if they are characters and A follows B in the ASCII character table. Otherwise returns true.
.Greater than or equal to	A >= B	Takes two operands of the same type. Returns false if they are floats and integers and A is mathematically less than B. Returns false if they are characters and A precedes B in the ASCII character table. Otherwise returns true.

3.2.6. Other Operators

Name	Notation	Description
Parentheses	({Some Expression})	Used for precedence control as what inside parentheses will be evaluated first.
Assign	A = {literal, variable, or an expression of the type of A};	Assigns the value of the right side of the = sign to the variable on the left side. If types don't match an exception is thrown. If the variable on the left already has a value an exception is thrown.

4. Data Types

Type	Syntax	Details
integer	int	32-bit, 2's complement
character	chr	8-bit, Ascii
float	flt	32-bit, IEEE 754
list	<type>	Single Type, Immutable
boolean	bln	True or False only
void	void	non-existent value
function	{paramType paramType … paramType **:** returnType} functionName **=** paramName paramName … paramName	First Class

4.1. Overview

There are four fairly standard data types that behave pretty much the same as they do in other languages. These types are integer, character, float and boolean. Additionally ALBS treats functions as first class data types in that they can returned and passed to other functions. Lists are integral enough to functional programming they are considered data types as well. All of the data types can be set to null.

4.2. Integer

Integers are represented with 32-bit memory chunks in 2's complement. This gives an maximum and minimum of 2,147,483,647 and -2,147,483,648 respectively. The standard arithmetic operators (+,-,/,*) can be applied to integers. Truncation is used to calculate the result of division if a non mathematical integer would be the mathematical result. For example 7/2 == 3 would return true.

```
int x = 2;
int y = 7/2; // value of y is 3
```

4.3. Character

We support the ASCII character set in using a byte of memory for each character. We use the system as ASCII where ASCII characters have a corresponding number and it is this number the the byte is set to (unsigned).

```
chr x = ‘a’;
chr y = ‘7’;
print x;//prints x
print y;//prints 7
int z = x + y; //adds 97 and 7. prints 104
```

4.4. Float

They are represented using 32 bits as according to the IEEE 754 standard. Each float must include a decimal point and have digits before, after, or both. In terms of regular expressions this can be represented as [‘0’-’9’]+ ‘.’ [‘0’-’9’]* | [‘0’-’9’]* ‘.’ [‘0’-’9’]+

```
flt x = 0.1;
flt y = .2;
flt z = 1.;
```

4.5. List

A list can be of type int, chr, bln, flt or a function. They are represented as <type>. Every element in the list must be of the same type.

```
<int> intList = [1,2];
```

A character list can be declared as:

```
<chr> list1 = [‘h’,’i’];
<chr> list2 = “hello”;
<chr> list3 = list2 + “ world”;
```

Lists are multidimensional, with the number of <> representing the list depth.

```
<<chr>> string1 =
[[‘h’, ’e’, ’l’, ’l’, ’o’], [’w’, ’o’, ’r’, ’l’, ’d’]];
//2 dimensional list of char


<<chr>> string2= [“hello”, “world”];
//2 dimensional list of char
```

A list of functions must have functions with the same signature. A list of functions can be passed values in order to return another list generated by applying the functions to the values passed.

```
{int int : int} add = x y [
     rtn x + y;
]
{int int : int} multiply = x y [
     rtn x * y;
]

<{int int : int}> listOfFunctions = [add, multiply];

<int> intList = listOfFunctions 1 2;
//passes 1,2 to all functions. Returns a list of integers

print intList;//prints [3,2]
```

4.6. Boolean

A boolean can be true or false.

```
bln x = true;
bln y = false;
print x & y; //prints false
```

4.7. Void

This represents a non-existent value. If a function returns nothing, then it returns void by default.

4.8. Function

Functions take in zero or more parameters and return a value of a predetermined data-type.

There is not a set keyword unlike the previous data types but instead the following convention is used:

```
{ paramType1 paramType2 …  paramTypeK : returnType}
     functionName =
     paramName1 paramName2 … paramNameK [
     //code
]
```

For example,

```
{int int : int} sampleFunction = x y [
```

```
        //code
    ]
```

Functions must be declared before they are called. Functions can be declared as:

```
{int int : int} sampleFunction;
```

5. Scope

Scope is controlled by curly brackets and square brackets from control structures and function definitions respectively.

On a function level scope is managed in the following way. Upon entering the function the parameters are at the outermost level of scope. Anything declared outside any control statements or inner function bodies are also at this level.

Each interior function and control structures creates a new scope one level lower than the level the interior function of control structure resides in and for each of these new levels of scope upon entering the function the parameters are at the outermost level of scope and, again, anything declared outside any control statements or inner function bodies are also at this level. This recursive nature can continue ad infinitum.

Variables residing in a scope level are not in any scope levels above the level the variable was declared in and are in every scope that is based of the current scope. These results in variables coming into into scope when they are declared and going out of scope when the matching closing parenthesis is found for opening parentheses most immediately preceding the variables declaration. When a function calls a function the scope of the caller is not inherited by the callee. All function not in another function are in all scopes.

Sample Code:

```
{int int: int} bar = a b [
    int z = a % b + 2;
    //ok because the only variables in this scope are a and b
    rtn z;
]

{int int: int} foo = x y [ //x and y are immediately in the scope

    int z = 0; //z at the same level of scope as x and y
    int w = bar x y; //w also at this level
```

```
      if w == 0 [
            int v = 200 + z;
            //legal because z is inherited from the outer scope.
      ]

      int v = w + bar x y;
      //ok because the v in the if will be removed from scope

      rtn v;
]
```

6. Control Flow

6.1. Program Entry and Exit

Entry into the program is accomplished by starting at the main function. This means that the program also starts with any command line arguments entered into the program when the program is executed. The main function is free to call other functions which can also call other functions or themselves. No function can call the main function. Functions can be declared outside the main (these are global functions), inside the main, or inside each other.

Here is what the main function can look like:

```
{void : void} helloWorldFunction = [

      <chr> hello = [‘h’, ’e’, ‘l’, ‘l’, ‘o’];
      <chr> space = [‘ ’];
      <chr> world = “world”;

      <chr> helloWorld = hello + space + world;

      print helloWorld; //[h,e,l,l,o, ,w,o,r,l,d]

      rtn void;
]

{<<chr>> : void} main = args [
      helloWorldFunction();
      rtn void;
]
```

This would result in hello world being printed.

6.2. Expression statements and blocks

Expression statements are assignments or function calls. Our language supports only expression statements, i.e. those statements that evaluate to a value. Statements are terminated by a semicolon.

Square braces [ ] are used to group statements together into blocks. There is no semicolon after a block ends with a square brace.

Sample code:

```
int x = 10; //this is an expression
print x; //this is an expression that returns void

if x==10 [ //this is a block
      print "equal";//prints a char list "hello"
]
```

6.3. If-Else

The if-else statement is used to express decisions.

Syntax:

```
if condition1 [
      statement1;
]
else [
      statement2;
]
```

Here, the else part is optional.
If the `condition1` evaluates to true, then the `statement1` is executed. If expression is false, and if there is an else part, `statement2` is executed instead.
The square brackets are mandatory and help avoid ambiguity.

6.4. Else-If

This is useful for writing a multi-way decision. For example:

```
if a==b[
      d=10;
]
else if b==c[
```

```
      d=20;
]
else[
      d=30;
]
```

The expressions are evaluated in order. If `a==b` is true, then the statements associated with it are executed, and this terminates the whole chain. Otherwise `b==c` is checked and if true the statements inside the brackets corresponding to the else if are executed. The last else statement is useful for checking the default case, however it can be omitted.

7. Punctuation

7.1. Semicolon

All expressions are terminated with the a semicolon.

```
int a = 25; //This is a variable declaration in ALBS.
```

7.2. Colon

The colon symbol is used to separate the parameter types and the return type in the function declaration.

```
{int int : int} add = x y [];
//The colon separates the two int parameters
//from the int return type of the add function.
```

7.3. Curly Braces

Curly braces are used for the enclosing of the parameter and return types in a function declaration.

```
{int int : int} add = x y [];
//The curly braces enclose the int parameters and return type.
```

7.4. White Space

White space is defined as spaces, tabs, newlines, and form feeds. It is ignored except to separate tokens.

To Illustrate:

```
If a == 0 [
                  Rtn 0;
]
```

Is equivalent to:

```
If a == 0 [ Rtn 0; ]
```

But not:

```
      Ifa==0
[Rtn0;]

Which would result in a parsing error.
```

7.5. Parentheses

Parenthese are used control the order of evaluation.

```
int x = 5 / 5 + 1; //evaluates to 2
int y = 5/(5 + 1); //evaluates to 0
```

7.6. Square Brackets

The square brackets must be used to enclose the function definition. They are also used to in if/if else/else constructs.

```
{int int: int} multiply = a b [
      if a == 0 [
            rtn 0;
      ]//if statement definition is enclosed with square brackets.
      else [
            int c = a - 1;
            rtn b + multiply c b;
      ]//else definition is enclosed with square brackets.
]//multiply definition is enclosed with square brackets.
```

Square braces are also used to enclose the values of a list.

```
<chr> hello = ['h','e','l','l','o'];
//The square braces enclose the chr values of the list hello.
```

8. Standard Library

8.1. I/O

Function name	Sample Code	Description
print	<int> intList = [1,2]; print intList; //[1,2]	Prints the argument passed to it.

input	<chr> myInput = input;	Takes input from the user as a character list. The user ends the input by the control+D command.

8.2. List Operations

Function name	Signature	Description
listLength	{<type>:int} listLength = myList [...]	Returns the length of the list myList as an integer value
head	{<type>: type} head = myList [...]	Returns the first element in a list <type>
tail	{<type>: <type>} tail = myList [...]	Returns a list containing all elements in a list <type> except the first one.
listGet	{<type>,int: type} listGet = myList, index [...]	Returns an element in a list myList at index
listAdd	{type,<type>,int: <type>} listAdd = newElement, myList, index[...]	Adds a new Element newElement to list myList at an optional index. If index is not specified, then the element gets added to the tail of the list by default. It returns a new list.
leftReduce	{{type,type:type},<type>: type} leftReduce = myFunction, myList[...]	It takes a function myFunction of {type,type:type}, applies myFunction to the first two elements in myList, and then applies myFunction to the result and third element in myList, and so on. It returns the final result of type type.
rightReduce	{{type,type:type},<type>: type} rightReduce = myFunction, myList[...]	It takes a function myFunction of {type,type:type}, applies myFunction to the first last elements in myList, and then applies myFunction to the result and third last element in myList, and so on. It returns the final result of type type.

Lists can be concatenated using the '+' operator.

8.2.1. Length of List

function name	`listLength`
arguments	`<type> myList`
return type	`type`
sample code	`<int> sampleList = [1,2,3,4,5];` `int len = listLength sampleList; //returns 5`
additional info	-

8.2.2. Head of List

function name	`head`
arguments	`<type> myList`
return type	`type`
sample code	`<int> sampleList = [1,2,3,4,5];` `int headVal = head sampleList; //returns 1`
additional info	Throws an exception if the list `myList` is empty

8.2.3. Tail of List

function name	`tail`
arguments	`<type> myList`
return type	`<type>`

sample code	`<int> sampleList = [1,2,3,4,5];`
	`<chr> tailList = tail sampleList;`
	`//returns [2,3,4,5]`

additional info	-

8.2.4. Get element at an index

function name	`listGet`
arguments	`<type> myList, Int index`
return type	`type`
sample code	`<chr> myString = ['a', 'b', 'c'];` `listGet 1 myString; //returns b`
additional info	-

8.2.5. Add an element to a list at an (optional) index

function name	`listAdd`
arguments	`type newElement, <type> myList, int index`
return type	`<type>`

sample code	`<chr> myString = ['a', 'b'];`
	`listAdd 'c' myString; //['a','b','c']`
	`listAdd 'c' myString 0;//['c','a','b']`

additional info	If `index` is not specified, then the element gets added to the tail of the list by default.

8.2.6. LeftReduce

function name	`leftReduce`
arguments	`{type,type:type},<type>`
return type	`<type>`
sample code	`<int> sampleList = [5,4,3,2,1];` `int sumAll = leftReduce + sampleList;//15`
additional info	If there are more than two arguments, then the user must specify the parameters in parenthesis to avoid ambiguity of how the function and list are built.

8.2.7. RightReduce

function name	`rightReduce`
arguments	`{type,type:type},<type>`
return type	`<type>`
sample code	`<int> sampleList = [5,4,3,2,1];` `int multiplyAll =rightReduce (funcThatReturnsAMultiplyFunc 'a') (funcThatReturnsAList sampleList);//120`
additional info	If there are more than two arguments, then the user must specify the parameters in parenthesis to avoid ambiguity of how the function and list are built.

9. Language Features

9.1. Single assignment

Variable can only be initialised when declared. They are immutable.

```
int x = 21;  // variable x is initialized and assigned
x = 25;      // Error: this is not allowed
int x = 25;  // Error: this is not allowed
int y = x;
x = x+1;     // Error: this is not allowed
```

9.2. First Class Functions

Functions can be passed as parameters and are assigned to a variable.

```
{int int : int} add = x y [
     rtn x + y; // explicit return statements
]

{{int int : int} int int : int} math = f m n [
//f is a first class function
     rtn f m, n;
]
int c = math add a b;
//parentheses not required for calling functions
```

9.3. Higher Order Functions

Functions can return functions.

```
{int : {int: int}} addX = x [
      {int : int} toReturn = z [
            rtn x + z;
      ]
      rtn toReturn;
]
{int: int} increment = addX 1;

increment 4;// returns 5
```

9.4. Closures

Functions can access variables that are in their scope, even if they are not explicitly passed to it.

```
{int int : int} addFunky = x y [

      {int : int} add = z [
            rtn x + z;//x is in scope
      ]
      rtn add y*2;
]
addFunky 2 3; //returns 8
```

9.5. Type casting

Casting from one type to another is allowed. Lists can also be casted. Refer to the conversion chart for casts for all valid casts.

```
chr myChar = (chr)2;//abc

<int> intList =  [1, 2, 3];
<chr> myChar = (<chr>) intList; //[1,2,3]

//concat
<chr> helloWorld = hello + space + world;
//[h,e,l,l,o, ,w,o,r,l,d]
```

10. References

“C Operator Precedence." *Cppreference.com*., 10 July 2015. Web.

Kernighan, Brian W., and Dennis M. Ritchie. *The C programming language*. Vol. 2. Englewood Cliffs: prentice-Hall, 1988.

Singh, Uday, et. al. “Superscript Language Reference Manual.” 2015. http://www.cs.columbia.edu/~sedwards/classes/2015/4115-fall/index.html