Arbol Programming Language: Language Reference Manual

February 24, 2021

Andrea Mary McCormick {amm2497}

Anthony Palmeira Nascimento {asp2199}

Derek Hui Zhang {dhz2104}

1. Introduction

In the realm of programming there is a plethora of languages that enable the implementation of trees. However, in many of those cases such implementations are not native to the language, which poses particular challenges to inexperienced programmers. Thus, the purpose of the ARBOL language is to abstract much of the logic behind the Binary Search Tree data structure in order to reduce the complexity and promote easy access to such useful functionality.

Our language – which is primary based upon the C programming language – includes built-in syntax in order to provide ease and usefulness for such Binary Search Trees by comprising node data structures as well as an element of self-balancing abstraction behind-the-scenes. The most common tree operations provided through our tree-specific syntax include: node declaration, child assignment, node dereferencing, and a get-child functionality that provides easy access to a given nodes' immediate children. With this new tree-specific syntax and the underlying layer of abstraction offered by ARBOL, users will be able to solve complex algorithmic BST problems without having to worry about properly implementing or operating on the tree data structure.

2. Lexical Conventions

An Arbol program is read by a parser, and input to the parser includes a stream of tokens, which are generated by the lexical analyzer. This section describes how the lexical analyzer breaks a file into tokens.

2.1 Comments

The character # introduces a single-line comment, which is terminated by the end of the line (‘\n’). The character “#*” introduces a multi-line comment, which is terminated with a corresponding “*#”.

2.2 Identifiers (Names)

An identifier includes some sequence of upper/lowercase letters, digits, and underscores ‘_’. The first character of an identifier must be alphabetic. Identifiers are case-sensitive.

Examples:

```
foo
foo_bar
tempChar
```

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise (i.e. as variable or function names):

```
function
return
if
else
for
```

```
while
continue
break
int
float
boolean
char
null
true
false
```

2.4. Literals

Literals are notations for constant values of some built-in types, and are as follows:

2.4.1 Integer Literal

An integer literal is a sequence of digits. An integer literal may begin with the optional unary operator minus sign (‘-’), which denotes a negative value, or it may begin with a digit. An integer literal must not begin with a zero if it contains more than one digit. All integer literals are interpreted as base-10 (decimal) numbers.

Examples:

```
89
-9
2537
0
```

2.4.2 Float Literal

A float literal consists of a signed integer part, a decimal part, or a fraction part. Both the integer part and decimal part of a float literal include a sequence of digits. Either the integer part, or the fraction part (but not both) may be missing.

Examples:

```
-2.46
```

```
62.0
.98
2.3
```

2.4.3 Boolean Literal

A Boolean literal has one of the two values: true; false.

2.4.4 Char Literal

A char literal consists of a single character enclosed in single-quotes (‘ ’). The following escape sequences exist in the Arbol Language and are preceded by a back-slash ‘\’ :

\n	Newline
\t	Tab
\\	Backslash

Examples:

```
‘b’
‘’
‘P’
‘9’
```

2.4.5 String Literal

A string literal consists of a sequence of characters surrounded by double quotes (“ ”). A string has the type: array-of-characters. The compiler places a null byte (\0) at the end of each string in order to specify the end.

Examples:

```
“sunshine”
“”
“968”
```

2.5 Punctuation

The following punctuation conventions are recognized within the Arbol Language and are as follows:

@	Create new node, get value of node
^	Get left or right child
-->, <--	Node assignment
( )	Expression precedence, conditional parameters, function arguments
[ ]	Value access (postfix expr)
{ }	Statement block
;	End of statement
,	Separate arguments in function declaration

2.6 Operators

The following operators are listed from top to bottom in order of precedence with their corresponding associativity:

**Top has the highest precedence: **

Precedence	**Associativity**	**Description**	**Operator**
1	Left-to-right	Child node	^
2	Left-to-right	Node dereference	@
3	Right-to-left	Logical NOT	!

4	Left-to-right	Multiplication, Division, Modulo (remainder)	* / %
5	Left-to-right	Addition, Subtraction	+ −
6	Left-to-right	Relational greater than, greater than or equal to, less than, less than or equal to	> ≥ < ≤
7	Left-to-right	Relational equal, not equal to	== !=
8	Left-to-right	Logical And	&&
9	Left-to-right	Logical Or	||
10	Left-to-right	Assignment equals, Node assignment	= ->

2.7 Whitespace

Instances of whitespace includes the following:

‘ ’	Blank character
\t	Tab character
\n	Newline character
\r	Carriage return

3. Types

3.1 Primitive Types

There are five primitive types:

`int`	32-bit integer
`float`	32-bit single precision floating point type
`char`	8-byte data type used to store ASCII characters
`boolean`	1-byte data type that is either ‘true’ or ‘false’
`void`	No value (usually used to signify the return type of a function that returns nothing)

3.2 Derived Types: `Node` Type

All nodes in a tree have the derived `Node` data type. This data type specifies how we enter data and what type of data we enter. Each `Node` data type includes a name, primitive type, value, and pointers to left/right children (if applicable).

This `Node` type is built into our language. Just as most other languages have built-in syntax for creating arrays, our language has built-in syntax for creating binary trees. All nodes in a tree must have the same type.

The above code creates an integer node named a and assigns the value of 5 to the node. Remember, a is not of type integer; a is a node type and thus the value cannot be accessed directly. In order to get the value of a, the dereference symbol @ must be used before the expression.

```
# A node can be created as follows:

      int @a -> 5;

      # Dereferencing node ‘a’ in order to access its value directly

      int @a -> 5;
      int x = @a + 3;
```

In the above code, the '@' dereference symbol in the last line indicates that we are getting the value of a, rather than just the node itself. The equal sign '=' is restricted to node pointer assignment. For instance, in the code below, the first line creates a new node of `int` type named b with a value of 5. However, the second line assigns the node a to point to the same node that was created in node b. The reference to a is being used; the second line does not create a new node.

```
# Create a new node b and assigning node a to point to the same node that
# was created in node b:

int @b -> 5;
int @a = b;
```

The "-->" operator assigns the node on the right side as the right child of the node on the left side of the operator. The "<--" operator assigns the node on the left side as the left child of the node on the right side of the operator. Note that the nodes on both sides must have the same type, or an error will be returned.

```
# Create three nodes; assign c as the left child of c and b as the right child of a.

int @a -> 1;
int @b -> 2;
int @c -> 3;
a --> b;
c <-- a;
```

The ^ operator is used to get the left or right child of the current node, depending on whether the ^ is the left or right side of the variable name. The ^ operator returns the node, not the value of the said node; to get the value, the node must also be dereferenced.

```
# Create three nodes; assign c as the left child of c and b as the right child of a.

int @a -> 1;
int @b -> 2;
a --> b;
int @c = a^ # get the right child of a
int d = @a^ # get the value of the right child of a
```

4. Expressions

4.1 Primary expressions

Primary expressions are literals, identifiers, or expressions in parentheses.

```
    primary_expression:
               literal
               identifier
               (expression)
```

4.1.1 Literal expressions

Literals are integer literals, float literals, boolean literals, character literals, and string literals.

4.1.2 Node expressions

A node expression allows declaring a node, value assignment for a given node, node re-assignment, child assignment, getting a child for a given node, and node dereferencing. See section 3.2 for additional details.

Node Declaration

```
type @ node
```

Value Assignment for Given Node

```
@ node -> expression
```

Node Re-Assignment (pointing to a new Node)

```
@ node = node
```

Child Node Assignment

```
node --> node
node <-- node
```

Get Child Node

```
node ^
^ node
```

Node Dereferencing

```
@ node
```

4.2 Operator expressions

The following are expressions that include arithmetic, relational, logical, or assignment operators:

Equality Operator

```
    expression == expression
```

Addition and Subtraction Operators

```
    expression + expression
    expression - expression
```

Multiplication, Division, and Modulo Operators

```
    expression * expression
    expression / expression
    expression % expression
```

Assignment Equals, Not-Equals Operators

```
    expression = expression
    expression != expression
```

Relational Operators

```
expression > expression
expression ≥ expression
expression < expression
expression ≤ expression
```

Logical And, Or, and Not Operators

```
expression && expression
expression || expression
! expression
```

4.3 Postfix expressions

Postfix expressions are primary expressions, ++ operator, -- operator, and expressions in brackets (denoting a reference to a particular value).

```
post_fix_expr:
          primary_expression
          ++
          --
          [expression]
```

4.4 Unary expressions

Unary expressions are postfix expressions and the unary minus operator (denoting a negative value).

```
unary_expression:
          post_fix_expr
          -
```

5. Functions

5.1 Function Declaration

A function in ARBOL can take zero or more arguments. In this example, `add` takes two parameters of type `int`. Notice that the type comes *before* the variable name.

```
function int add(int x, int y) {
     return x + y;
}
```

5.2 Built-in Functions

ARBOL supports a multitude of tree related built-in functions for operating on nodes. Just a few of the most prominent ones are shown below:

height(root)	Returns the maximum height of the tree or subtree
preorder(root)	Returns an array with an preorder traversal of the tree or subtree
inorder(root)	Returns an array with an inorder traversal of the tree or subtree
postorder(root)	Returns an array with an postorder traversal of the tree or subtree
inlevel(root)	Returns an array with an in-level (BFS) traversal of the tree or subtree
getlevels(root)	Returns a mxn matrix containing all levels in the tree or subtree
validate(root)	Returns a boolean. True if a tree with type integer is a valid BST. False otherwise.

6. Statements

Statements are executed sequentially.

6.1 Expression Statements

Expression statements are mostly used as assignments or function calls.

```
expr;
```

6.2 Conditional Statements

Conditional statements are executed as follows (note that the else statement is optional):

```
if (x == 5) {
     return true;
} else {
     return false;
}
```

6.3 While Statements

While statements can be written as follows::

```
while(x < 5) {
     x = x + 1;
}
```

6.4 For Statements

For statements can be executed as follows:

```
for (int x = 0; x < 20; x++) {
    print(“Hi”);
}
```

6.5 Break Statements

Like in C, break statements can be used to exit out of control statements early:

```
for (int x = 0; x < 20; x++) {
    if (x == 5) {
        break;
    }
}
```

6.6 Continue Statement

Like in C, the continue keyword is used to indicate that we should continue to the next iteration of the control statement, as opposed to exiting the control statement early.

```
arr = [1,2,3,4,5];

for (int i = 0; i < len(arr); i++) {
    if (arr[i] == 3) {
        continue;
    }
    print(arr[i])
}
```

When the above code is executed, it produces the following result

```
1
2
4
5
```

6.7 Return Statement

Always, only one value can be returned from a function. A return statement with an expression shall not appear in a function whose return type is void. A return statement without an expression shall only appear in a function whose return type is void.

6.8 Null Statement

A null statement (consisting of just a semicolon) performs no operations.