TreeSL

THE TREE SPECIFIC LANGUAGE

GROUP – 14

Rishi Manoj Poluri - CS21BTECH11045

Ritvik Sai Chippa - CS21BTECH11054

Nishanth Bhoomi - CS21BTECH11040

Dhanisetty Sai Anvitha - CS23BTNSK11001

"Embrace the branches of knowledge, for in the forest of Binary Trees and BSTs, we find the roots of data structures and the leaves of algorithmic beauty." 🌲🌟

INDEX

I. INTRODUCTION

Motivation for Designing:

The inspiration behind developing the Binary Trees and Binary Search Trees. The basic importance of binary trees and binary search trees in computer science and data structures is the foundation of this DSL. Various algorithms, data storage, and search operations all depend on these tree structures in one way or another. Because traditional programming languages often lack particular features to effectively operate with these structures, manual coding is required, which takes time. Our DSL aims to fill this gap by providing a tool for working with binary trees and BSTs, improving the development cycle, and ensuring accurate and efficient implementations.

Goal:

By creating DSL for binary trees and BSTs that simplifies and speeds up their usage, we mainly aim to empower developers. A set of high level concepts and functions designed clearly for these data structures are what this DSL tries to provide. By doing this, it reduces coding complexity, improves code quality, and speeds up the creation of algorithms and software applications that depend on binary trees and BSTs. We hope to encourage best practices for using tree-based data structures and make it simple for developers to maximize their potential.

II. Lexical Conventions:

2.1 Comments:

Both single and multiline comments are supported in this DSL.

All tokens after a # symbol on a line is considered to be part of a comment and are ignored by

the compiler. The multi comments are implemented by the #* The comment is written here *#

2.2 Whitespace

Whitespace, including tabs, spaces, newlines and comments, will be ignored, except in places where spaces are required for the separation of tokens. Tabs/indentation cannot be used to define the scope in this DSL.

2.3 Reserved Keywords

The following words are reserved keywords in the DSL and they cannot be used as regular identifiers.

If, else, return, switch, case, break, print,continue, null,int, long, float, double, char, string, bool, void, for, while, true, false, Node, BTree, BSTree.

2.4 Identifiers

Here are the constraints on the identifiers:

i. Identifiers do not start with the digits

ii. Special characters are not allowed except the '_' ,'$' and '~'.

iii. Reserved keywords cannot be identifiers.

iv. Identifiers can contain alphabets and are case sensitive.

Examples:

(i) Allowed : _FOo, _foo, fo~$2
(ii) Not Allowed: 1foo, fo3o@, int

2.5 Punctuations

, ; ' " ( ) [ ] { } .

2.6 Literals

The literals are the character ,floating point numbers , integers and string constants.

III. DataTypes:

3.1 Primitive Data types

- int
- float
- char
- string
- bool
- void
- double
- long

3.2 Non-Primitive Data types

- Node
  Node has three access members: left ,right and val.
- BSTree
- BTree

The member functions for BSTree and BTree are mentioned in the section 7.

IV. Operators

Purpose	Symbol	Associativity	Valid Operands
Parentheses for grouping of operations	( )	left to right	int, float, double,char, string, Node, BTree,BSTree
Square parentheses for grouping elements	[ ]	left to right	int
Curly parentheses for initializing elements	{ }	left to right	int, float, char, null, Node,double,long
Member access operator	.	left to right	Node,BSTree,BTree
Unary negation	!	right to left	bool
Increment	++	right to left	int, float, double,long
Decrement	--	right to left	int, float, double,long
Modulo	%	left to right	int
Multiplication	*	left to right	int, float, double,long
Division	/	left to right	int, float, double,long
Addition	+	left to right	int, float, double,long,,BTree
Subtraction	-	left to right	int, float, double,long,BTree
Relational Operators	<= < >= >	left to right	all datatypes except BTree,BSTree,Node
	== !=	left to right	all datatypes
Logical Operators	AND OR	left to right	bool

Purpose	Symbol	Associativity	Valid Operands
Bitwise Operators	& | ^	left to right	int, float, double,bool
Assignment operators	=	right to left	LHS is a variable, RHS is an expression, LHS and RHS have the same type

V. Declarations

A program consists of various entities such as variables, functions, types. Each of these entities must be declared before they can be used.

Example-1:

```
int foo(int a)
{
 return a + 42;
}

void main()
{
Node<int> a = {1, null, null};
Node<int> b = {2, null, null};
BTree<int> c = {a, null, b};
foo(2);
string str1 = "float is ";
}
```

5.1 Variable Declarations

For primitive datatypes the format for the declaration is:
Datatype id;
Datatype id1, id2, …;

For non-primitive datatypes the format for declaration is:
non-primitive<primitive> id;
non-primitive<primitive> id1, id2, …;

Examples:

```
int a,g;
```

float foo;
Node<float> fla;
BTree<int> tree;
BSTree<int> Tree;
bool a1;
char r;
string str;

5.2 Declaration Scope

In this DSL, the scope is defined by using { }. The identifier introduced by a declaration is valid only within the scope where the declaration occurs. Variables declared in global scope must have unique identifiers. The same identifier cannot be used to refer to more than one entity in a given local scope.

5.3 Array Declaration

Only one dimensional traditional array is allowed (one-dimensional array in C)
Allowed datatypes for array are primitive like int, float, etc and Node is non-primitive.
Example:

```
Int arr[10];
Node<int> arr[10];
```

5.4 Function Declaration

The syntax for function declaration:

```
return_type functionName(datatype_1 arg_1, datatype_2 arg_2, ...);
```

A function may or may not have arguments but the return type is a must (return type void can be
used if none is required).

5.5 Initializers:

When an object is declared, its init-declarator may specify an initial value for the identifier being
declared. The initializer is preceded by = and is either an expression or a list of initializers
nested in braces.

5.5.1 Types of Initializations:

5.5.1.1 Default initialization:

When variables are declared but not initialized, they are initialized with default values for primitive data types (similar to those in C++):

Example:

int a; (default 0)
float a; (default 0.0 for floating points)

For non-primitive datatypes:

Example:

Node<int> a; (value=0, left=null, right=null)
BTree<int> b; ({Node<int>}- rootnode, with default Node)

5.5.1.2 Direct initialization:

Syntax for primitive:
Datatype id = literal (of same type)

For non-primitive:
Node<int> a = {1, null, node1}
BTree<int> b = {node1, null, node2, …} (These nodes on RHS are corresponding to each position in tree in level order)

5.5.1.3 Copy initialization:

It is the initialization of one variable using another variable. This kind of initialization can be used for all supported data types.

Example:

int a = b;
b=c;

VI. Statements

6.1 Expression Statements:

Statements are executed for their effect and do not have values. Each expression statement includes one expression, which is usually an assignment or a function call. In this DSL, every expression is followed by a semicolon ";" .

Syntax:
expression ;

6.2 Compound Statements:

A compound statement is a sequence of statements enclosed by braces as follow:

Syntax:

```
{
statement1;
statement2;
statement3;
statement4;
}
```

If a variable is declared in a compound statement, then the scope of this variable is limited to this statement.

6.3 Control Flow:

6.3.1 Conditionals:

If statements consist of a predicate (an expression) and a series of statements. The series of statements are evaluated if the predicate evaluates to True. If the predicate evaluates to False, either the program continues or an optional else clause is executed.

Predicate: predicate here is the “expression” which evaluates only to a boolean value.

Syntax:

```
if (predicate) {
Statements;
}
else if (predicate){
Statements;
}
else {
Statements;
}
```

6.3.2 Loop Statements:

We have two loops, for loop and while loop.

6.3.2.1 FOR-LOOP:

The for loop has three parts, namely:

- **Initialization:** This part is executed once at the beginning of the loop. It is typically used to initialize a loop control variable.
- **Predicate:** predicate here is the “expression” which evaluates only to a boolean value.
- **Update:** This part is executed after each iteration of the loop. It is usually used to update the loop control variable.

Syntax:

```
for (initialization; predicate; update) {
   // code to be executed repeatedly
}
```

Example:

```
int a=9;
for(int i=0;i<65;i++){
        a=a+7;
}
```

6.3.2.2 WHILE-LOOP:

Predicate: predicate here is the “expression” which evaluates only to a boolean value.

Syntax:

```
while (predicate) {
   // code to be executed repeatedly
}
```

Example:

```
int q=56;
while(q<0){
q=q/2;
}
```

6.3.2.3 Break statement:

The break statement ends the loop immediately when it is encountered.

Example:

```
int x=9;
while(x<0)
{
if(x==2)
x=x-2;
```

else
break;
}

6.3.2.4 Continue statement:

The continue statement skips the current iteration of the loop and continues with the next iteration.
Examples :

```
Int x=2;
while(x< 0)
{
if(x==2)
continue;
x=x+2;
}
```

7. Built-In functions:

- Functions like print
- Traversals : Inorder,Postorder,Preorder,DFS,BFS
- Node Search
- Merge(Merges two Trees)
- Node Deletion
- Node Insertion

NOTE : OUTPUT AND INPUT SPECIFICATIONS ARE YET TO BE MODIFIED OR ADDED. FEW OTHER CHANGES CAN BE MADE ACCORDING TO THE IMPLEMENTATION REQUIREMENTS