CMPT 432 - Spring 2024 - Dr. Labouseur

Project Three

Collin Drake
Collin.Drake1@Marist.edu

April 8, 2024

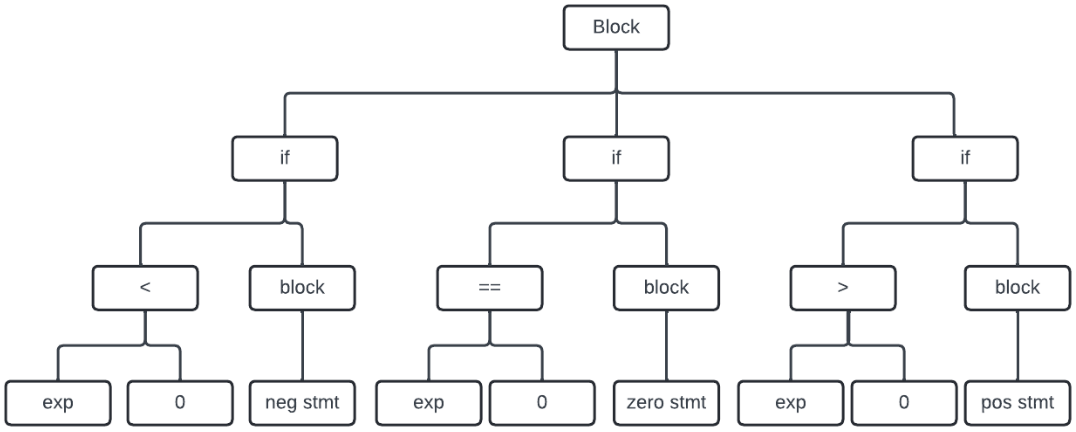
Lab 5

Crafting a Compiler Exercises:

Exercise: 9.2

Show the AST you would use for this construct. Revise the semantic analysis, reachability, and throws visitors for if statements to handle the signtest.

signtest (exp) {
neg: stmts
zero: stmts
pos: stmts
}



Lab 6

Crafting a Compiler Exercises:

Exercise: 8.1

The two data structures most commonly used to implement symbol tables in production compilers are binary search trees and hash tables. What are the advantages and disadvantages of using each of these data structures for symbol tables?

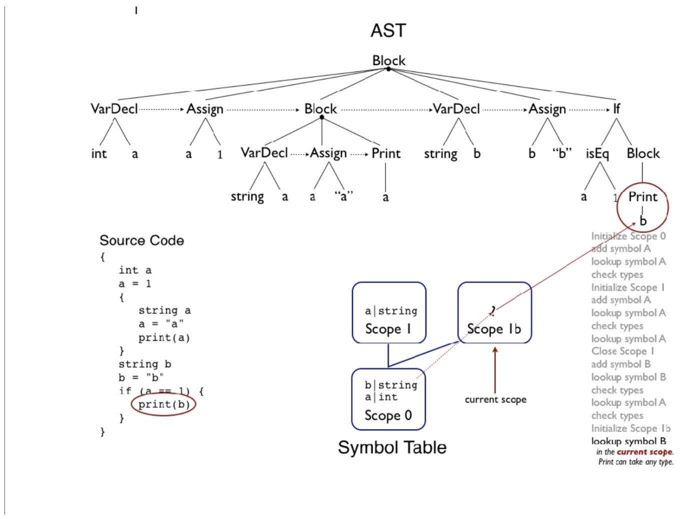
In terms of using either of these data structures for the implementation of a symbol table, they both have their advantages and disadvantages. Binary search trees are straightforward to implement because they are well-known and they also make it simple to manage and handle scope-related operations. However, some downsides to binary search trees are that they have very slow lookup times and are difficult to keep balanced. On the other hand, hash tables have fast and consistent lookup times, but depending on the type of collision handling you choose this can increase the complexity and decrease its available size.

Lab 7

Alan Exercises:

Describe in detail what is happening in the diagram below.

In regards to the diagram below, a compiler's semantic analyzer is determining if the variable 'b' is a valid variable that exists within the program's scope 1b. After checking the current scope for the variable 'b' and not finding it, the semantic analyzer will proceed to check the parent scope until it either finds the variable 'b' or determines that this was an undeclared variable and throws an error. In this case, the semantic analyzer was able to find the variable 'b' within the parent scope 0, therefore, validating the print statement.



Source Code

```
{
   int a
   a = 1
   {
      string a
      a = "a"
      print(a)
   }
   string b
   b = "b"
   if (a == 1) {
      print(b)
   }
}
```

Initialize Scope 0
add symbol A
lookup symbol A
check types
Initialize Scope 1
add symbol A
lookup symbol A
check types
lookup symbol A
Close Scope 1
add symbol B
lookup symbol B
check types
lookup symbol A
check types
Initialize Scope 1b
lookup symbol B
*in the **current scope**.*
Print can take any type.

Project 3

Semantic Analysis

In the Compilation process, Semantic Analysis is the third step. In this step, we accept the Concrete Syntax Tree produced by the Parser and produce an Abstract Syntax Tree, containing only the "good stuff" from the CST. Later we traverse the newly created AST and use it to construct a Symbol Table. While we produce these representations we are focused on the meaning of the source code and enforcing the scope and type of variables.

To test my Semantic Analyzer I used the test cases below:

Test Case 1

```
        {
          boolean d
          d  = (true == (true == false))
          if((2+2==4)!=("hi"=="hello")){
            print(d)
          }
          {
            int a
            a = 1 + 5
            {
              string c
              c = "hello"
              if("hello" == "hi"){
                print(c)
                if(1 == 5){
                  print(a)
                }
              }
            }
          }
        }$
```

Test Case 2

```
        {
          boolean d
          d = (true == (true == false))
          print(d)
          {
            int a
            int b
            string c
            a = 1
            b = 5
            c = "hello"
            if(a == b){
              print(a)
              {
                print(b)
              }
            }
          }
          {
            int a
            a = 7
          }
        }$
```

Test Case 3

```
{
  int a
  int b
  b = 1
  a = 5 + 4 + b
  {
    print("i")
    print("think")
  }
  if(a == b){
    print(a)
    while(1 == (1 == 1)){
      print("no")
    }
  }
}$
```

Test Case 4

```
{
  int x
  string y
  boolean z
  {
    x = 7
    y = "mr␣villain"
    while(y != "hello"){
      z = false
      print(z)
    }
    string z
    z = "its␣my␣day␣off"
  }
}$
```

1 Appendix

Symbol.java

```
/*
    Symbol file
    Creates symbols to be added to the Symbol Table
    Used in Semantic Analysis
*/

//The symbol class!
public class Symbol{
    String name;    //the name of the symbol
    String type;    //the symbols type
    boolean isINIT; //is this symbol initialized?
    boolean isUsed; //is this symbol used?
    int scope;  //what is the symbols scope?
    String line;    //the line in the source code where the token is located

    //Symbol constructor -- creates a symbol and initializes its variables
    public Symbol(String name, String type, int scope, String line){
        this.name = name;
        this.type = type;
        this.isINIT = false;
        this.isUsed = false;
        this.scope = scope;
        this.line = line;
    }
}
```

SymbolTableNode.java

```
/*
    Symbol table node file
    Creates nodes for the symbol table
*/

//import arraylist and hashtable
import java.util.ArrayList;
import java.util.Hashtable;

//The symbol table node class!
public class SymbolTableNode {
    int scope;  //the scope within the program
    SymbolTableNode parent; //pointer to parent node
    Hashtable<String, Symbol> symbols;    //hashtable of symbols
    ArrayList<SymbolTableNode> children;   //list of pointers to child nodes

    //Symbol table node constructor -- initializes all variables
    public SymbolTableNode(int scope){
        this.scope = scope;
        this.parent = null;
        this.symbols = new Hashtable<>();
        this.children = new ArrayList<>();
    }
}
```

ASTNode.java

```
        /*
            AST Node file
            Creates AST Nodes to be used in the Semantic Analyzer and Symbol Table
            These nodes contain the tokens created by the lexer
        */

        //import arraylist
        import java.util.ArrayList;

        //The AST Node class!
        public class ASTNode {
            String name;    //the name of the node
            Token token;    //if the node contains a token...
            ASTNode parent;    //pointer to the nodes parent
            ArrayList<ASTNode> children;   //list of pointers to child nodes

            //AST Node constructor -- creates a node and initializes its variables
            public ASTNode(String label, Token token){
                this.name = label;
                this.token = token;
                this.parent = null;
                this.children = new ArrayList<>();
            }
        }
```

References

Links

Below are the resources I have used to create simple, readable, and beautiful code.

- Helped me when attempting to output my symbol table: stackoverflow
- Assisted with learning about hashtables and their implementation: geeksforgeeks
- The semantic analysis slides: labouseur.com
- Test cases and output checking: Arnell Compiler
- Output and scope checking: Nightingale Compiler