Paul Kummer

Dr. Hanku Lee

Compilers CSIS 455-01

11/12/2021

Lab 10: Type Checking / Symbol Table / Grammar Checking

## Purpose of Lab

This lab's purpose is to check the grammar of within the code that is to be compiled and check whether types are compatible. First, whenever IdentifierNodes are encountered, the symbol table is looked up to see if the node has been declared. If an IdentifierNode is attempted to be assigned a value, but hasn't been declared, an error will occur. Also, when an IdentifierNode is being declared, the symbol table is updated with the new node and it is assigned a type. The types saved to IdentifierNodes are important so when an expression is encountered, the nodes can be compared for compatibility. Whenever nodes are not of compatible types, the compiler will exit and display an error message.

## Example Code

Environment

To store variables within the code and create variables confined to a local environment, the Env.java class is created. It is a hashtable of types <Token, IdentifierNode> that also has a link to a previous hashtable. By having a link to the previous hashtable, and environment can be restored to what it formerly was after leaving a block statement. Since the Env class is used in other parts of the compiler, it is stored in the current working directory as the Main.java.

*Env.java*

```
8   public class Env
9   {
10      private Hashtable table;
11      protected Env prev;
12
13      public Env ()
14      {
15          this.table = new Hashtable();
16      }
17
18      public Env (Env n)
19      {
20          table = new Hashtable();
21          prev = n;
22      }
23
24      public void put(Token w, IdentifierNode id)
25      {
26          table.put(w, id);
27      }
28
29      public IdentifierNode get(Token w)
30      {
31          if(w != null)
32          {
33              for (Env e = this; e != null; e = e.prev)
34              {
35                  IdentifierNode found = (IdentifierNode)(e.table.get(w));
36                  if (found != null)
37                  {
38                      return found;
39                  }
40              }
41          }
42
43          return null;
44      }
45  }
```

Lexer

The lexer phase has been modified to save the reserved words to the symbol table. It may not be necessary but may be used in the future for preventing users from assigning a value to a reserved keyword. Also, an end-of-file Word is added.

*Lexer.java*

```
12  public class Lexer
13  {
14      public int line = 1 ;
15      private char peek = ' ' ;
16      private File file = new File("input.txt");
17      private BufferedReader br;
18      //private Hashtable<String, Token> words = new Hashtable<String, Token>() ;
19      public Env reserved = new Env();
20      public Hashtable words = new Hashtable() ;
```

Allows the reserved words from the lexer to be passed into other phases.

```
73      void reserve (Word w)
74      {
75          words.put(w.lexeme, w) ;
76          reserved.put(w, new IdentifierNode(w));
77      }
```

Saves the reserved work into the lexer's environment.

*Word.java / Tag.java / Lexer.java*

```
66          //misc
67          reserve(Word.Eof);
```

```
53      //misc
54      public final static int EOF     = 0xFFFF; // end of file
```

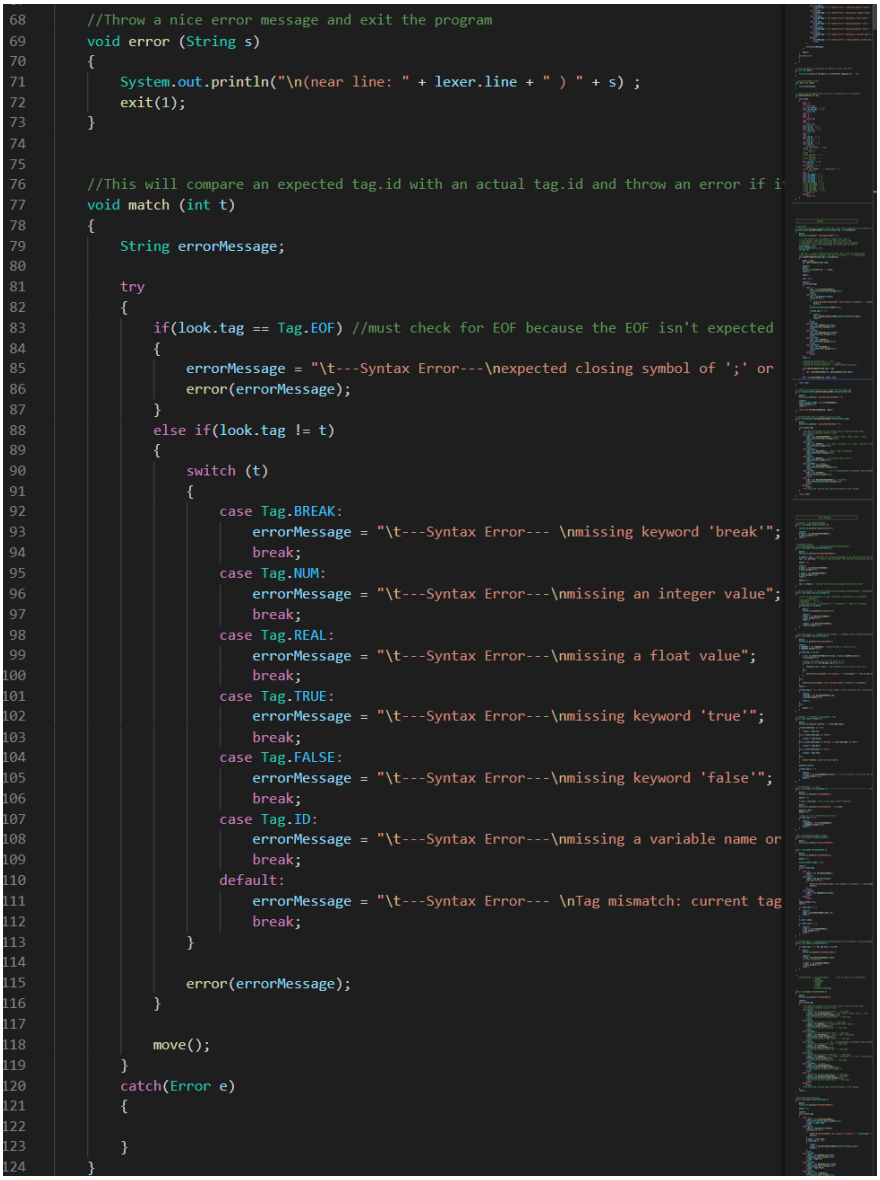
```
48    //eof
49    public static final Word Eof       = new Word ("eof",          Tag.EOF);
```

An end-of-file word has been added to detect when a noncharacter is encountered.

Parser

The parsing phase has undergone many changes and now does more grammar checking along with storing values to the symbol table. Now whenever an IdentifierNode is declared, it is saved to the symbol table with its type. If an IdentifierNode is encountered outside of a declaration statement, the IdentifierNode is recalled from the symbol table using the current token. If the IdentifierNode is recalled as null, then an error is thrown, and the program closes because a value is attempting to be stored to an identifier that has not yet been declared. Also, the symbol table is added to every block statement, so each block statement can have its own environment of local variables.

*Parser.java*



```
68      //Throw a nice error message and exit the program
69      void error (String s)
70      {
71          System.out.println("\n(near line: " + lexer.line + " ) " + s) ;
72          exit(1);
73      }
74
75
76      //This will compare an expected tag.id with an actual tag.id and throw an error if i
77      void match (int t)
78      {
79          String errorMessage;
80
81          try
82          {
83              if(look.tag == Tag.EOF) //must check for EOF because the EOF isn't expected
84              {
85                  errorMessage = "\t---Syntax Error---\nexpected closing symbol of ';' or
86                  error(errorMessage);
87              }
88              else if(look.tag != t)
89              {
90                  switch (t)
91                  {
92                      case Tag.BREAK:
93                          errorMessage = "\t---Syntax Error--- \nmissing keyword 'break'";
94                          break;
95                      case Tag.NUM:
96                          errorMessage = "\t---Syntax Error---\nmissing an integer value";
97                          break;
98                      case Tag.REAL:
99                          errorMessage = "\t---Syntax Error---\nmissing a float value";
100                         break;
101                     case Tag.TRUE:
102                         errorMessage = "\t---Syntax Error---\nmissing keyword 'true'";
103                         break;
104                     case Tag.FALSE:
105                         errorMessage = "\t---Syntax Error---\nmissing keyword 'false'";
106                         break;
107                     case Tag.ID:
108                         errorMessage = "\t---Syntax Error---\nmissing a variable name or
109                         break;
110                     default:
111                         errorMessage = "\t---Syntax Error--- \nTag mismatch: current tag
112                         break;
113                 }
114
115                 error(errorMessage);
116             }
117
118             move();
119         }
120         catch(Error e)
121         {
122
123         }
124     }
```

The match has been updated to give more detailed error messages for common grammar errors. When an error occurs the error method is called.

*Parser.java cont.*

```
107     // exit the program nicely
108     void exit (int value)
109     {
110         System.exit(value);
111     }
```

This new method will exit the program when an error happens.

```
452         if(look.tag == Tag.ID)
453         {
454             n.id = new IdentifierNode((Word)look, (Type)(n.typeNode.basic));
455             n.id.accept(this);
456 
457             //update the symbol table for the token of n.id
458             if(n.id.w != null && top.get(n.id.w) == null)
459             {
460                 top.put(n.id.w, n.id); // the hashtable does not allow a null entry
461             }
462             else
463             {
464                 error("DeclarationNode: The variable [ " + (Word)look + " ] HAS already
465             }
466         }
467         else
468         {
469             error("DeclarationNode: only a variable name or location is accepted");
470         }
```

When a variable is declared, the symbol table is checked and updated if the variable doesn't exist.

```
930         n.left = top.get((Word)look);
931         if(n.left == null)
932         {
933             error("AssignmentNode(LHS): the variable or location [ " + (Word)look + " ] 
934             exit(1);
935         }
936         ((IdentifierNode)n.left).accept(this);
```

Like the declaration, the symbol table is searched for a variable. If the variable exists, the value can be retrieved from the table. Otherwise, an error will occur.

Type Checker

The newly introduced phase compares node types in expressions or assignments to determine whether they are compatible. If the types are not compatible, the compiler will throw and error message and exit the program.

*TypeChecker.java*

```
66      public Type compareTypes(Type lhs, Type rhs)
67      {
68          if(lhs == null && rhs == null)
69          {
70              error("Missing types");
71          }
72          else if(lhs != null && rhs == null)
73          {
74              println("LHS's Type: " + rhs);
75              return lhs;
76          }
77          else if(lhs == null && rhs != null)
78          {
79              println("RHS's Type: " + rhs);
80              return rhs;
81          }
82          else if(lhs == Type.Float)
83          {
84              if(rhs != Type.Int && rhs != Type.Float)
85              {
86                  error("Type mismatch: the types must be numeric (bad Type) --> " + rhs);
87              }
88
89              println("LHS's Type: " + lhs);
90              println("RHS's Type: " + rhs);
91              return Type.Float;
92          }
93          else if(lhs == Type.Int)
94          {
95              if(rhs != Type.Int)
96              {
97                  error("Type mismatch: the types must be Integers (bad Type) --> " + rhs);
98              }
99
100             println("LHS's Type: " + lhs);
101             println("RHS's Type: " + rhs);
102             return Type.Int;
103         }
104         else if(lhs == Type.Bool)
105         {
106             if(rhs != Type.Bool)
107             {
108                 error("Type mismatch: the types must be Boolean (bad Type) --> " + rhs);
109             }
110
111             println("LHS's Type: " + lhs);
112             println("RHS's Type: " + rhs);
113             return Type.Bool;
114         }
115         else
116         {
117             System.out.println("Could not compare types");
118         }
119         return null;
120     }
```

This method will compare two different types for compatibility. If the types are incompatible, an error message is displayed, and the program exits. For instance, a float and an integer are compatible types. However, when a float and an integer are used together, the derived type will be of lower precision and be an integer. The derived type is returned by the method, which can be used to assign a type to a binary expression.

*TypeChecker.java cont.*

```java
public void visit (AssignmentNode n)
{
    println("AssignmentNode");
    Type leftType = null;
    Type rightType = null;

    if(n.left instanceof IdentifierNode)
    {
        n.left.accept(this);
        leftType = ((IdentifierNode)n.left).type;
        println("LHS's Type: " + leftType);
    }
    else
    {
        error("Could not get LHS type");
    }

    if(n.right != null)
    {
        n.right.accept(this);
        if(n.right instanceof NumNode)
        {
            rightType = Type.Int;
        }
        else if(n.right instanceof RealNode)
        {
            rightType = Type.Float;
        }
        else if(n.right instanceof TrueNode || n.right instanceof FalseNode)
        {
            rightType = Type.Bool;
        }
        else if(n.right instanceof IdentifierNode)
        {
            rightType = ((IdentifierNode)n.right).type;
        }
        else if(n.right instanceof BinaryNode)
        {
            rightType = ((BinaryNode)n.right).type;
        }
        else
        {
            error("Could not get RHS type");
        }
    }

    Type assignType = compareTypes(leftType, rightType);
}
```

The visit method to an AssignmentNode will compare the type of the left-hand-side, which must be an IdentifierNode, with the right-hand-side expression's type. If the right-hand-side is a leaf of the AST, the type can be determined right away. Otherwise, the type has to be derived from an ExpressionNode. Once both types have been determined, the compareTypes() method is used to check compatibility.

*TypeChecker.java cont.*

```
307     public void visit (BinaryNode n)
308     {
309         println("BinaryNode: " + n.op);
310         n.type = null;
311         Type leftType = null;
312         Type rightType = null;
313 
314         if(n.left != null)
315         {
316             n.left.accept(this);
317 
318             if(n.left instanceof NumNode)
319             {
320                 leftType = Type.Int;
321             }
322             else if(n.left instanceof RealNode)
323             {
324                 leftType = Type.Float;
325             }
326             else if(n.left instanceof TrueNode || n.right instanceof FalseNode)
327             {
328                 leftType = Type.Bool;
329             }
330             else if(n.left instanceof IdentifierNode)
331             {
332                 leftType = ((IdentifierNode)n.left).type;
333             }
334             else if(n.left instanceof BinaryNode)
335             {
336                 leftType = ((BinaryNode)n.left).type;
337             }
338             else
339             {
340                 error("Could not get LHS type");
341             }
342             n.type = leftType;
343         }
344 
345         if(n.right != null)
346         {
347             n.right.accept(this);
348             if(n.right instanceof NumNode)
349             {
350                 rightType = Type.Int;
351             }
352             else if(n.right instanceof RealNode)
353             {
354                 rightType = Type.Float;
355             }
356             else if(n.right instanceof TrueNode || n.right instanceof FalseNode)
357             {
358                 rightType = Type.Bool;
359             }
360             else if(n.right instanceof IdentifierNode)
361             {
362                 rightType = ((IdentifierNode)n.right).type;
363             }
364             else if(n.right instanceof BinaryNode)
365             {
366                 rightType = ((BinaryNode)n.right).type;
367             }
368             else
369             {
370                 error("Could not get RHS type");
371             }
372             n.type = compareTypes(leftType, rightType);
373         }
374     }
```

The vist method for a binary expression behaves similar to the visit method to the assignment node. However, one important difference is that the binary expression gets a value assigned to it, which is used in the

Unparser

The tree printer remains mostly unchanged and still prints off the AST of all the nodes. The Unparser has been changed to have the TypeChecker passed into it, otherwise it it unchanged as well.

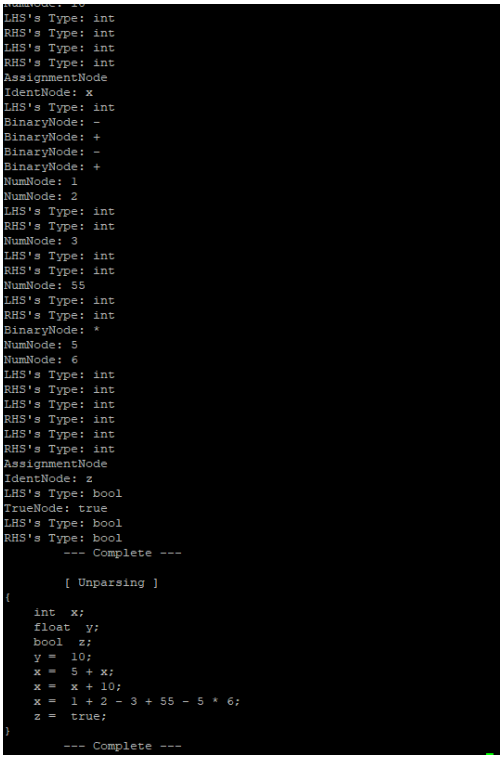
Visitor

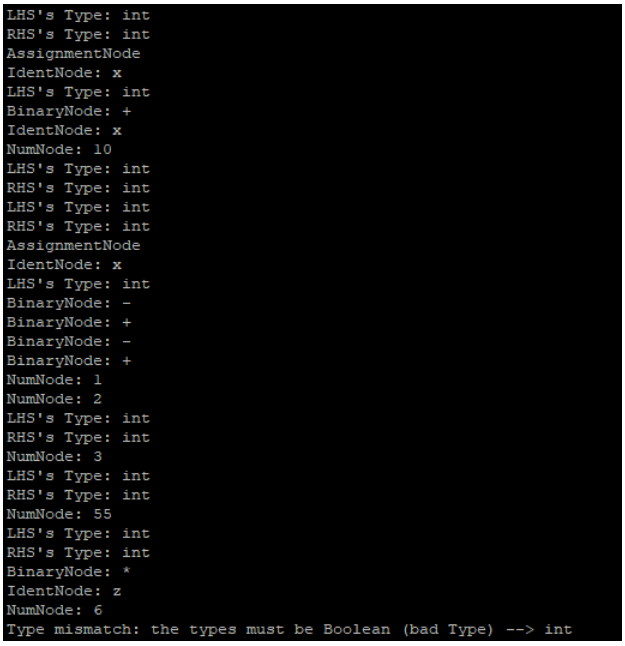
No changes have been made to the ASTVisitor.

AST

The binary node and expression node now have types assigned to them, so it can be accessed in the typechecker.

**Execution**




When there is a type mismatch, the type checker will output an error message. In the example above and on the right, an error occurs because the type of variable “z” is declared to be a Boolean and the NumNode of value 6 and type Integer is attempted to be assigned to “z”. The types of Bool and Int are incompatible and do not make logical sense. therefore, the error occurs. The example on the left shows correct type matching. The assignment of variable “y”, a float, to the NumNode of “10”, and integer, is allowed because an integer can be coerced into a float without any precision being lost.

**Conclusion**

This lab taught me how to do grammar checking and type checking. The intricacies of getting the types to be checked correctly required some methodical thinking. It isn’t too complicated, but it’s easy to not understand what is happening if one step is missed. The biggest problem I encountered was from a bug I somehow introduced into the parseBinaryExpresion() method. I had a while loop outside of another while loop when it was supposed to be an inner while loop. This error was very difficult to trace. I believe that I have a good base of a compiler now and will begin to remove vestigial code from previous labs and assignments to make everything more organized and easier to detect errors. This lab also helped with understanding how to correctly implement the symbol table.