Analysis, Design, and Construction of a SIMPLE Interpreter

Advanced Topics Computer Science
Compilers and Interpreters

Jonathan Lee
Mr. Page
21 November 2014

INTRODUCTION

The purpose of this document is to analyze, design, and construct a SIMPLE Interpreter given by a set of grammar. Though interpreters receive the same input, the actual process of interpretation can vary in both efficiency and complexity depending on the grammar. This document analyzes the efficiencies and limitations of a particular interpreter grammar and identifies a design for an interpreter that parses according to an adjusted variation of the given grammar and produces the intended output.

DESIGN OF THE INTERPRETER

```
Program →      Statement P
P →            Program | ∊
Statement →    display Expression St1
               | assign id = Expression
               | while Expression do Program end
               | if Expression then Program St2
St1 →          read id | ∊
Expression →   Expression relop AddExpr
               | AddExpr
AddExpr →      AddExpr + MultExpr
               | AddExpr – MultExpr
               | MultExpr
MultExpr →     MultExpr * NegExpr
               | MultExpr / NegExpr
               | NegExpr
NegExpr →      -Value
               | Value
Value →        id | number | (Expression)
```

Figure 1: The given grammar

LIMITATIONS AND SEMANTIC ISSUES

The given grammar in its most basic form, shown in *Figure 1* above, is both logical and intuitive. A program is a compilation of one or more statements, expressions account for order of operations, and terminal tokens are defined as operators, integers, identifiers and grammar-specific keywords. Though intuitive, the grammar poses several major syntactical limitations.

The foremost limitation is the grammar's adjustment for order of operations. The given grammar accounts for the correct order of operations for the parsing of the input. As opposed to evaluating expressions in the order they appear, the interpreter will evaluate them according to the standard order of operations. Typically, descending precedence offers a simple solution to the issue of order of operations when parsing an arithmetic expression. Descending precedence is the reason for the separation of types of expressions into `AddExpr`, `MultExpr`, and `NegExpr` in that particular order. The effects of descending precedence can be seen in *Figure 2* below.

No precedence	Descending Precedence
Input	Input
`3 + 4 * 5`	`3 + 4 * 5`
Output	Output
`35`	`23`

Figure 2: Non-precedence and precedence comparison

Since the non-precedence method evaluates expressions in the order that it scans them, its output is inconsistent when multiplicative operations are mixed with additive expressions. Via descending precedence, multiplicative operations always end up closer to the leaves of the abstract syntax tree; thus the parser evaluates these operations before the additive operations. However, the initial specification of the grammar combined with the nature of the parser's one-token look-ahead causes an endless recursive loop. For instance, observe the definition of the nonterminal `AddExpr` in *Figure 1*. In general, it is perfectly logical; however, the first non-terminal that it encounters is itself, `AddExpr`. Thus, this second `AddExpr` defines *itself* with another `AddExpr`. This process would recur infinitely. The only method to maintain this exact grammar would be to have the parser parse backwards by finding the last additive operator while constructing an expression and then retracing its steps to evaluate backwards. This "look back" solution would prove to be highly inefficient, as the parser would have to encounter the same tokens more than once and then store them. The solution to this left recursive issue involves transforming the grammar and is described in depth in the next section.

Another consequence involving expressions is the fact that conditional expressions are grouped directly as expressions rather than having their own grammatical definition, which would logically be two different expressions, surrounding a relative operator. Conditional statements will thus have little expectancy for the return value of the condition, as the return value could theoretically be any integer rather than a constrained `1` or `0` or a `true` or `false` value. Similarly, considering the way the grammar is defined, a condition could contain multiple relative operators thus essentially acting as a expression operator that outputs only `1` or `0`.

Lastly the program has no actual termination keyword or symbol. Since `Program` does not a have a finite limit to the number of statements it may contain, the only method of termination is to receive no token input from the `Scanner`. While this is not functionally deleterious to the interpretation, it does add another level of ambiguity.

GRAMMAR TRANSFORMATIONS AND REASONING

A recapitulation of the syntactical analyses is as follows: left recursion causes endlessly recursive loops; expression-condition confluence causes ambiguity between two components of different applications; lack of specific termination token causes minor ambiguity.

The justification for the transformation of the left recursion issue is presented in the above section. In short, left recursive parsing of input would cause the parser to recursively loop within the first expression token because of its one-token look-ahead nature. Thus, the efficient solution to this issue would be to transform the grammar with the use of elimination of left recursion in order to avoid an endless loop. A comparison of an arbitrary grammar is given below in *Figure 3*. It should be noted that the grammars are logically equivalent and accurately describe the same output but their methodologies are different.

Left Recursion	Elimination of Left Recursion
$S \rightarrow S\alpha \mid \beta$	$S \rightarrow \beta S'$
	$S' \rightarrow \alpha S' \mid \epsilon$

Figure 3: Left recursion (original) and elimination comparison

As shown, a parser with one-step look-ahead limitation would only efficiently work with elimination of left recursion. Elimination entails another non-terminal that defines itself as either an empty set or a specific token type followed by a reference to itself. This completely avoids the issue of a recursive definition with no break point.

Although a transformation of the expression-condition confluence would make the program more reliable and robust, the transformation is not functionally necessary. Thus this design does not transform the theory of the expression-condition confluence; however, it should be noted that such a transformation could indeed correct a semantic ambiguity.

Again the lack of a program-ending token would not functionally change the parsing of the program; it would only add constraints to any given program's construction and thus does not require grammatical transformation.

DEMONSTRATION OF GRAMMAR TRANSFORMATIONS

As previously stated, the specific grammar transformation to eliminate left recursion does not functionally alter the original grammar. Below is a simplified example of the parsing of the input `3 * 8 + 2` in *Figure 4*.

Left Recursion	Elimination of Left Recursion
1. `Expression` 2. `AddExpr + MultExpr` 3. `MultExpr + MultExpr` 4. `MultExpr * NegExpr + MultExpr` 5. `NegExpr * NegExpr + NegExpr` 6. **`number`** `*` **`number`** `+` **`number`**	1. `Expression` 2. `AddExpr whileExpression` 3. `MultExpr whileAddExpr whileExpression` 4. `NegExpr whileMultExpr whileAddExpr whileExpression` 5. **`number`** `* NegExpr whileMultExpr whileAddExpr whileExpression` 6. **`number`** `*` **`number`** `+ MultExpr whileAddExpr whileExpression` 7. **`number`** `*` **`number`** `+` **`number`**

Figure 4: Demonstration showing that both grammar varieties yield the same result

DECOMPOSITION AND SERVICES OF OBJECTS

Executions and evaluations of the SIMPLE interpreter are carried out by individual objects that each represents a piece of the abstract syntax tree. Those objects are divided into statements, which are executed, and expressions, which are evaluated. Other types of objects exist with various functions.

- `Statement` – generic abstract class that includes an implementable method for executions
  - `Program` – class that represents the most universal statement. `Program` objects are `Statement` objects that are made up of a list of `Statement` objects all to be executed.
  - `Display` – class that describes the display function to output an expression to the console and potentially assign a new identifier based on input.
  - `Read` – class that represents a statement that only follows a `Display Statement`. This class is responsible for assigning input from the console to a given variable
  - `Assignment` – class that handles the assignment of expressions to specific identifiers via the environment.
  - `While` – class that handles while loops given an expression and series of statements in the form of a program.
  - `If` – class that describes an `if` statement with a conditional expression and a following program. `If` objects may also have following else programs.
- `Expression` – abstract class that includes an implementable method for evaluations and, by default, evaluates as a condition.
  - `Number` – class the represents basic numbers with integer values. `Numbers` are evaluated for exactly the integer they represent.

    - `Variable` – class that handles variables by storing an identifier and referencing the environment to determine the value of the identifier.
    - `BinOp` – expression that represents a binary operator that holds two expressions and an operator. When evaluated, the object performs the operation based on the given operator, which could be a relative operator to determine the value. `BinOp` objects represent conditions as well as `1` and `0`.
- Environment – the variable environment that acts as a reference source when the program needs to access the values of any used variables.

TEST STRATEGY

The test for the interpreter simply examines whether or not the functions implicitly listed in the grammar actually work as expected. For example, the entire program should be able to handle an unspecified number of `Statement` objects; `Display` should print to the console and should also be followed by a `Read` statement to ensure that both work jointly. The resulting variable from the `Read` object should be displayed as well; an `Assignment` function should be carried out and then the variable should be printed to ensure that the assignment does indeed assign the value to the variable; a `While` statement should be run conveniently with the assigned variable with a condition that can change; an `If` statement with an `else` component should be executed. Each component should display an `Expression` with all types of operations: add, subtract, multiply, divide.

SOURCE CODE

Parser.java

```java
package parser;

import java.io.File;
import java.io.FileInputStream;
import java.util.ArrayList;
import java.util.List;

import environment.Environment;
import scanner.Scanner;

import ast.Assignment;
import ast.BinOp;
import ast.Display;
import ast.If;
import ast.Program;
import ast.Read;
import ast.Statement;
import ast.Expression;
import ast.Number;
import ast.Variable;
import ast.While;

/**
```

```
 * Parser class designed to recognize grammar of tokens delivered by the
 * scanner. Tokens are used to construct representative objects which
 * are then arranged in the form of an abstract syntax tree for execution
 * and evaluation as a whole program.
 *
 * Some components of the grammar have been defined as continuation components
 * These components have a potential e option where there is nothing to return.
 * in this case, e could entail a null token in which case, the program may throw a null pointer exception.
 * so in each of these programs, the null point must be handled.
 *
 *
 * The general non-optimized grammar is given by
 * Program      ->       Statement P
 * P            ->       Program | e
 * Statement    ->       display Expression St1
 *                      | assign id = Expression
 *                      | while Expression do Program end
 *                      | if Expression then Program St2
 * St1          ->       read id | e
 * St2          ->       end | else Program end
 * Expression   ->       Expression relop AddExpr
 *                      | AddExpr
 * AddExpr      ->       AddExpr + MultExpr
 *                      | AddExpr - MultExpr
 *                      | MultExpr
 * MultExpr     ->       MultExpr * NegExpr
 *                      | MultExpr / NegExpr
 *                      | NegExpr
 * NegExpr      ->       -Value
 *                      | Value
 * Value        ->       id | number | (Expression)
 *
 * This given set of grammar poses multiple limitations and semantic issues

 * that are described thoroughly in the accompanying document.
 * Accounting for such limitations and semantic issues, the optimized
 * grammar can be designed that appropriately eliminates left recusion and left factoring
 * and other issues that arise.
 * Program      ->       Statement P
 * P            ->       Program | e
 * Statement    ->       display Expression St1
 *                      | assign id = Expression
 *                      | while Expression do Program end
 *                      | if Expression then Program St2
 * St1          ->       read id | e
 * St2          ->       end | else Program end
 * Expression   ->       AddExpr Expression'
 * Expression'  ->       relop AddExpr Expression' | e
 * AddExpr      ->       MultExpr AddExpr'
 * AddExpr'     ->       - MultExpr AddExpr' | + MultExpr AddExpr' | e
```

```java
 * MultExpr     ->       NegExpr MultExpr'
 * MultExpr'    ->       * NegExpr MultExpr' | / NegExpr MultExpr' | e
 * NegExpr      ->       - Value | Value
 * Value        ->       (Expression) | id | num
 * Program can be defined as a statement followed by 0 or more statements
 *
 * @author Jonathan Lee
 * @vesion 10 Nov 2014
 *
 */
public class Parser
{
    /**
     * method: main
     * main method used to initialize the parser obejct
     * with a given text file name for parsing
     * @param args      system arguments default
     */
    public static void main(String[] args)
    {
        Parser parser = new Parser("file.txt");
    }

    /**
     * var: token
     * current token that the parser is examining from scanner
     */
    private String token;

    /**
     * var: scanner
     * scanner object that returns tokens in sequence from input
     */
    private Scanner scanner;
    /**
     * method: Parser
     * usage: new Parser()
     * creates new hashmap. gets file for scanner
     * this method parses the input until it reaches a . indicating
     * the end of input. Exceptions can be thrown from this, so it
     * is surrounded in a try-catch block to catch any
     * one of the two methods that modifies token
     * the parsed statement is then executed by the parser once its ast is created
     * precondition: file exists
     * postcondition: input parsed using left factoring parse methods
     */
    public Parser(String fileName)
    {
        //map = new HashMap<String, Integer>();
        File file = new File("src/files/" + fileName);
        try
        {
            System.out.println("file name: " + file.getName());
            scanner = new Scanner(new FileInputStream(file));
            token = scanner.nextToken();
            Environment env = new Environment();
            Program program = parseProgram();
            program.exec(env);

        }
        catch(Exception e)      //exception would be a file not found exception or an exception in nextToken, in that case, shut down
        {
            e.printStackTrace();
            System.exit(0);
        }
    }

    /**
     * method: parseProgram
     * method: program.parseProgram
     * method parses the program by compiling a list of
     * statements that are in the program and returning the object
     * @return  returned program with a list of parsed statements
     * postcondition: input advanced
     */
    public Program parseProgram() throws Exception
    {
        List<Statement> statements = new ArrayList<Statement>();
        Statement stmt = parseStatement();
        while(stmt != null)
        {
            statements.add(stmt);
            stmt = parseStatement();
        }
        return new Program(statements);
    }

    /**
     * method: parseStatement
     * usage: program.parseSTatemnet
     * parses the statement by first determining the type of statement
     * based on the first given token and then handling it accordingly
     * see the above grammar for specifics about how each type of statement
     * is handled.
     * @return statement    parsed statement returned as an object
     * precondition: token is not null
     * postcondition: input advanced
     */
    public Statement parseStatement() throws Exception
    {
        Statement statement = null;
        if(token.equals("") ||              //these are a list of keyword tokens that would yield no statement
                token.equals("end") ||      //this if statement his here specifically so that if the parser
```

```java
                token.equals("else"))       //counters these variables, it does not throw an exception
        {

        }
        else if(token.equals("display"))
        {
            eat("display");
            Expression expr = parseExpression();
            Read read = parseRead();
            statement = new Display(expr, read);
        }
        else if(token.equals("assign"))
        {
            eat("assign");
            String id = token;
            eat(token);
            eat("=");
            statement = new Assignment(id, parseExpression());
        }
        else if(token.equals("while"))
        {
            eat("while");
            Expression condition = parseExpression();
            eat("do");
            Program program = parseProgram();
            eat("end");
            statement = new While(condition, program);
        }
        else if(token.equals("if"))
        {
            eat("if");
            Expression condition = parseExpression();
            eat("then");
            Program program = parseProgram();
            Program altProgram = null;
            if(token.equals("else"))
            {
                eat("else");
                altProgram =  parseProgram();
                eat("end");
            }
            else
                eat("end");
            statement = new If(condition, program, altProgram);
        }
        else
        {
            throw new Exception("Unexpected token: " + token);
        }
        return statement;
    }

    /**
```

```
         * method: parseExpression
         * usage: program.parseExpression
         * parse expression is handled as other types of expressions
         * using the method of left recursion elimination
         * expressions are also handled as conditions with relative operators in this
         * case
         * @return  expression represented as a condition with a relative operator and expressions in a binop
         *          could also just be a general arithmetic expression without relative operators
         * @throws Exception
         */
        public Expression parseExpression() throws Exception
        {
            Expression exp1 = parseAddExpr();
            while(token.equals("=")         ||
                    token.equals("<>")    ||
                    token.equals(">")    ||
                    token.equals("<")    ||
                    token.equals("<=")   ||
                    token.equals(">="))
            {
                String op = token;
                eat(op);
                Expression exp2 = parseAddExpr();
                exp1 = new BinOp(op, exp1, exp2);
            }
            return exp1;
        }

        /**
         * method: parseRead()
         * usage: program.parseRead()
         * continuation component (see above)
         * parses a potential read element that follows a display by getting
         * the identifier name
         * @return  read object with identifier name
         * @throws Exception
         */
        public Read parseRead() throws Exception
        {
            if(token.equals("read"))
            {
                eat("read");
                Read read = new Read(token);
                eat(token);
                return read;

            }
            else
                return null;
        }
```

```java

    /**
     * method: parseAddExpr()
     * method that parses an add or subtract operations
     * same way as parseExpression does with left recursive elimination
     * helps with descending precedence
     * @return          returns an add expression as a binop or a mult expression
     * @throws Exception
     */
    public Expression parseAddExpr() throws Exception
    {
        Expression exp1 = parseMultExpr();
        while(token.equals("+") || token.equals("-"))
        {
            String op = token;
            eat(op);
            Expression exp2 = parseMultExpr();
            exp1 = new BinOp(op, exp1, exp2);
        }
        return exp1;
    }

    /**
     * method: parseMultExpr()
     * method that parses an mult or divs operations
     * same way as parseExpression does with left recursive elimination
     * helps with descending precedence
     * @return          returns an mult expression as a binop or a neg expression
     * @throws Exception
     */
    public Expression parseMultExpr() throws Exception
    {
        Expression exp1 = parseNegExpr();
        while(token.equals("*") || token.equals("/"))
        {
            String op = token;
            eat(op);
            Expression exp2 = parseNegExpr();
            exp1 = new BinOp(op, exp1, exp2);
        }
        return exp1;
    }

    /**
     * method: parseNegExpr
     * parses potentially negative expressions by determining if the next term
     * leads with a negative sign. it then constructs a binop with a multiplcation
     * operation with -1 for the opposite sign value
     * if no negative sign, then just passes expression on to value
     * @return
```

```
         * @throws Exception
         */
        public Expression parseNegExpr() throws Exception
        {
            if(token.equals("-"))
            {
                eat("-");
                Expression neg = new Number(-1);
                return new BinOp("*", parseNegExpr(), neg);
            }
            else
                return parseValue();
        }
        /**
         * method: parseValue
         * parses a value by determining the type of value
         * could be a term with parenthesis in which case the
         * value is an expression that is parsed yet again
         * could be an identifier in which case it is a variable
         * could be just a number in which case the number is returned.
         * @return
         * @throws Exception
         */
        public Expression parseValue() throws Exception
        {
            Expression exp = null;
            if (token.equals("("))
            {
                eat("(");
                exp = parseExpression();
                eat(")");
            }
            else if(scanner.isLetter(token.charAt(0)))
                exp = parseVariable();
            else
                exp = parseNumber();
            return exp;
        }

        /**
         * method: parseVariable
         * usage: program.parseVariable()
         * parses the variable by constructing it as an expression
         * with the given identifier as the token
         * postcondition: advances down input
         * @return  returns the constructed variable with the given name
         * @throws Exception described in eat()
         */
        public Expression parseVariable() throws Exception
        {
            Variable var = new Variable(token);
            eat(token);
            return var;
        }
```

```java

        /**
         * method: parseNumber
         * usage: program.parseNumber()
         * parses the number by constructing a number object
         * with the given integer as its value using
         * the parse integer value to get the int value from the string token
         * postcondition: advances down input
         * @return the constructed number with its value
         * @throws Exception described in eat()
         */
        public Expression parseNumber() throws Exception
        {
            return new Number(parseInteger());
        }

        /**
         * method: parseInteger
         * usage: program.parseInteger()
         * this method parses the given number from the token string form to an integer form
         * basically asks as a format converter
         * calls the eat method to advance down input
         * precondition: token is a number
         * postcondition: number token has been eaten
         * @return  integer value of the current token
         * @throws Exception described in eat()
         */
        private int parseInteger() throws Exception
        {
            int num = Integer.parseInt(token);
            eat(token); //this could throw an exception
            return num;
        }

        /**
         * method: eat
         * usage: program.eat(string)
         * "eats" the current token by reassigning token to the next
         * token returned by the scanner. takes an expected token to match with current
         * token to make sure that they are consistent (thus synchronized for the next token)
         * exception thrown if they are not (program quits because there is nothing you can do)
         * one of the two methods that modify token
         * @param expected      the expected token to be received
         * postcondition: token set to the next token given by the scanner
         * @throws IllegalArgumentException     if expected and current are not matching
         */
        private void eat(String expected) throws Exception
        {
            if(expected.equals(token))
                token = scanner.nextToken();
            else
                throw new IllegalArgumentException("Expected token: '" + exp
ected + "'. Received token: '" + token + "'.");
        }
    }
```

Environment.java

```java
package environment;

import java.util.HashMap;
/**
 * Environment class that represents the environment in which
 * procedures and variables are referenced. hashmaps are used
 * as the core of the environment
 * environments essentially represent scopes of the program.
 * each scope is independent of the other
 * this environment object only represents the global scope
 *
 * @author Jonathan Lee
 * @version 3 Nov 2014
 *
 *
 *
 */
public class Environment
{
    private HashMap<String, Integer> map;

    /**
     * method: Environment
     * constructor method for the class which just initializes
     * the hashmap objects as one of string-integer and one of
     * string-statement
     * postcondition: hashmaps initialized
     */
    public Environment()
    {
        map = new HashMap<String, Integer>();
    }

    /**
     * method: setVariable
     * usage: program.setVariable(variable, value)
     * puts the key and object values into the hasmp
     * specifically for integer references
     * so this corresponds to an integer value
     * @param variable      variable name for the integer
     * @param value         the integer value of the variable
     * postcondition: declares the variable by setting name and value corre
spondence
     */
    public void setVariable(String variable, int value)
    {
        map.put(variable, value);
    }

    /**
     * method: getVariable
     * usage: program.getVaariable(variable)
     * getter method using a variable parameter
     * to get the value from the environment
     * that it references, for integers
     * @param variable      variable name of the integer
     * @return              the integer with given variable name
     */
    public int getVariable(String variable)
    {
        return map.get(variable);
    }
}
```

Statement.java

```java
package ast;

import environment.Environment;

/**
 * Abstract statement class extended by sub classes
 * in order to execute various statements in different ways
 * @author Jonathan Lee
 * @version 9 Oct 2014
 *
 */
public abstract class Statement
{
    /**
     * executions that can be implemented in different ways depending on ob
ject
     * @param env   environment used for variable definitions
     */
    public abstract void exec(Environment env);
}
```

Expression.java

```java
package ast;

import environment.Environment;

/**
 * Abstract expression class extended by sub classes
 * in order to evaluate various statements in different ways
 * @author Jonathan Lee
 * @version 9 Oct 2014
 *
 */
```

```java
public abstract class Expression
{
    /**
     * evaluations that can be implemented in different ways depending on object
     * @param env   environment used for variable definitions
     */
    public abstract int eval(Environment env);
}
```

Program.java

```java
package ast;

import java.util.List;

import environment.Environment;

/**
 * Program class that represents a series of executable statements.
 * Programs could represent the entire interpreted program of the input
 * or any general series of statements such as that following a then keyword
 * of an if statement
 * @author Jonathan Lee
 * @version 18 Nov 2014
 *
 */
public class Program extends Statement
{
    private List<Statement> statements;

    /**
     * method: Program
     * usage: new Program(statements)
     * constructor method that takes a series of statments
     * right off the bat to prepare for execution
     * @param statements    list of statements to be executed by the program
     * postcondition: instance field of statements set
     */
    public Program(List<Statement> statements)
    {
        this.statements = statements;
    }

    @Override
    /**
     * method: exec
     * usage: program.exec
     * execution method that simply executes all the statements in the given
     * array list. nothing more.
     * @param env   environment used to pass on to other statements within
     *              the program
     * postcondition: statements in instance variable list executed
     */
    public void exec(Environment env)
    {
        for (int i = 0; i < statements.size(); i++)
            statements.get(i).exec(env);
    }
}
```

BinOp.java

```
package ast;

import environment.Environment;

/**
 * BinOp class which represents an expression
 * involving multiple expressions joined by an operator
 * for a specific mathematical operation
 * @author Jonathan Lee
 * @version 9 Oct2014
 *
 */
public class BinOp extends Expression
{
    private String op;
    private Expression exp1;
    private Expression exp2;

    /**
     * method: BinOp
     * constructor method that takes in two expressions
     * and one operator to construct the overall expression
     * represented by individual instance field
     * @param op    operator that determines the operation
     * @param exp1  first part of the expression operation
     * @param exp2  second part of the expression operation
     */
    public BinOp(String op, Expression exp1, Expression exp2)
    {
        this.op = op;
        this.exp1 = exp1;
        this.exp2 = exp2;
    }

    @Override
    /**
     * method: eval
     * usage: program.eval(env)
     * method that evaluates the expression by determining
     * the type of operator which is then used to carry out
     * the operation using the two expressions to join them
     * returns -1 if no operator recognized.
     * also recognizes options for relative operators
     * which test for true or false usually in conditions in the form of 1
and 0 respectively
     * conditions are grouped with general expressions
     * in this grammar. 1 or 0 is retruned for true and false respectively
     * @param env   environment used for variable definitions
     * @return  the value of the expression
     */
    public int eval(Environment env)
    {
        if(op.equals("="))
            if (exp1.eval(env) == exp2.eval(env))
                return 1;
            else
                return 0;
        else if(op.equals("<>"))
            if (exp1.eval(env) != exp2.eval(env))
                return 1;
            else
                return 0;
        else if(op.equals("<"))
            if (exp1.eval(env) < exp2.eval(env))
                return 1;
            else
                return 0;
        else if(op.equals(">"))
            if (exp1.eval(env) > exp2.eval(env))
                return 1;
            else
                return 0;
        else if(op.equals("<="))
            if (exp1.eval(env) <= exp2.eval(env))
                return 1;
            else
                return 0;
        else if(op.equals(">="))
            if (exp1.eval(env) >= exp2.eval(env))
                return 1;
            else
                return 0;
        else if(op.equals("+"))
            return exp1.eval(env) + exp2.eval(env);
        else if(op.equals("-"))
            return exp1.eval(env) - exp2.eval(env);
        else if(op.equals("*"))
            return exp1.eval(env) * exp2.eval(env);
        else if(op.equals("/"))
            return exp1.eval(env) / exp2.eval(env);
        else
            return -1;
    }
}
```

Display.java

```
package ast;
```

```java

import environment.Environment;

/**
 * Display class that describes the display function to output an
 * expression to the console and potentially assign a new identifier
 * based on input.
 * displays are given in the form of an expression
 * and reading values is done by the read object
 * reads are not required, so it may be null
 * @author Jonathan Lee
 * @version 18 Nov 2014
 *
 */

public class Display extends Statement
{
    private Read read;
    private Expression expr;

    /**
     * method: Display
     * constructor method that acts
     * as a setter for both the expression
     * and the read statement that follows
     * note that this constructor assumes a read
     * value is present (defaults to other constructor if not)
     * @param expr      the epxression to be displayed
     * @param read      the statement that reads the input for a specific assignment
     * postcondition: instance field set
     */
    public Display(Expression expr, Read read)
    {
        this.expr = expr;
        this.read = read;
    }

    /**
     * method: Display
     * variation of the constructor that
     * only takes an expression
     * indicating that there is no read after the display
     * see other constructor's description
     * @param expr  expression to be displayed
     * postcondition: instance field set
     */
    public Display(Expression expr)
    {
        this.expr = expr;
        this.read = null;
    }

    @Override
    /**
     * method: exec
     * method that prints out the evaluation of the expression
     * and then attempts to read for input with a read statement
     * @param env   environment for variable reference
     * postcondition: displayed expression and potentially read input
     */
    public void exec(Environment env)
    {
        System.out.println(expr.eval(env));
        if(read != null)
            read.exec(env);
    }
}
```

Assignment.java

```java
package ast;

import environment.Environment;

/**
 * assignemtn class which represents
 * a statement that assigns a variable to
 * an expression when evaluated
 * @author Jonathan Lee
 * @version 9 Oct2014
 *
 */
public class Assignment extends Statement
{
    private String var;
    private Expression exp;

    /**
     * method: assignment
     * constructor method that takes in the varaible name
     * and the expression that it will be assigned to
     * @param var       variable name
     * @param exp       expression that variable is associated with
     * postcondition: instance field set
     */
    public Assignment(String var, Expression exp)
    {
        this.var = var;
        this.exp = exp;
    }

    @Override
    /**
     * method: exec
     * usage: program.exec()
     * executes the statement by setting hte variable
     * name to the expression value in the env
     * variable for future reference.
     * @param env   environment used for variable definitions
     * postcondition: env set with new key-value
     */
    public void exec(Environment env)
    {
        env.setVariable(var, exp.eval(env));
    }
}
```

Read.java

```java
package ast;
import java.util.Scanner;
import environment.Environment;
/**
 * Read class statement that represents the process of taking input from
 * the console. this is supposed to be preceded by a display object
 * since the display on takes read after itself. Reads involve taking
 * an identifier to begin with and then prompting input from the console
 * the input is then assigned to the identifier
 * @author Jonathan Lee
 * @version 20 Nov 2014
 *
 */
public class Read extends Statement
{
    private String id;
    /**
     * method: Read
     * constructor method for read class
     * that is just a setter method
     * sets the id to the instance field
     * @param id    name of the variable to be assigned
     */
    public Read(String id)
    {
        this.id = id;
    }

    @Override
    /**
     * method: exec
     * usage: program.exec(env)
     * executes the read method by prompting for input from
     * the console using a scanner and then creating an assignment
     * object that assigns the input to the given variable.
     */
    public void exec(Environment env)
    {
        Scanner in = new Scanner(System.in);
        int x = in.nextInt();
        Assignment assign = new Assignment(id, new Number(x));
        assign.exec(env);
    }
}
```

While.java

```java
package ast;

import environment.Environment;

/**
 * while class which represents a while statement as a subclass of statement
 * this is an executable object that evaluates its condition
 * and determines whether or not to continue running its program
 * @author Jonathan Lee
 * @version 20 Nov 2014
 *
 */
public class While extends Statement
{
    private Expression cond;
    private Program program;

    /**
     * method: While
     * constructor method for the while class
     * which acts as a setter for the while condition
     * and the program to be executed
     * @param cond          condition expression that acts as the determinant for whether the while will be executed
     * @param program       program that is to be executed should the condition be true
     */
    public While(Expression cond, Program program)
    {
        this.cond = cond;
        this.program = program;
    }

    @Override
    /**
     * method: exec
     * execution method that evaluates the condition expression
     * to determine if it is 1 or 0 which then determines whether to
     * run the program continuously.
     * @param env       environment that handles references to variables
     */
    public void exec(Environment env)
    {
        while(cond.eval(env) > 0)
            program.exec(env);
    }
}
```

If.java

```java
package ast;

```

```java
import environment.Environment;

/**
 * class that represents an if statement (meaning that
 * it extends the statement abstract class
 * this class takes in a condition and then
 * a statement to execute if that condition is true
 * when an if statement is executed, it tests
 * for the value of the condition and then
 * determines whether or not to execute the given stmt
 * @author Jonathan Lee
 * @version 13 Oct 2014
 *
 *
 */
public class If extends Statement
{
    private Expression cond;
    private Program program;
    private Program altProgram;


    /**
     * method: If
     * constructor method used to set the values
     * of the instance fields cond and program and altProgram
     * @param cond  condition that the if statement tests before executing statement
     * @param program   program that the if statement executes if cond is true
     * @param altProgram    alternate program that acts as the actions of the else statement
     * postcondition: instance field set
     */
    public If(Expression cond, Program program, Program altProgram)
    {
        this.cond = cond;
        this.program = program;
        this.altProgram = altProgram;
    }

    /**
     * method: exec
     * usage: program.exec(env)
     * method that executes the if statement
     * by evaluating the condition and determining whether
     * it is true or false. if true, it executes its given program
     * if false it does nothing or executes the altProgram as long as altProgram
     * is not null
     * @param env   environment used for variable reference
     * postcondition: cond evaluated, maybe stmt executed
     */
    public void exec(Environment env)
    {
        if(cond.eval(env) > 0)
            program.exec(env);
        else if(altProgram != null)
            altProgram.exec(env);
    }
}
```

Variable.java

```java
package ast;

import environment.Environment;

/**
 * Variable class the represents a variable
 * which corresponds to a value in the environment
 * just contains the reference as a string object
 * for the name
 * @author Jonathan Lee
 * @version 9 Oct2014
 *
 */
public class Variable extends Expression
{
    private String name;

    /**
     * method: Variable
     * constructor method that takes in a name
     * to set for the instance field
     * @param name  name that will be set for reference
     * postcondition: name set
     */
    public Variable(String name)
    {
        this.name = name;
    }

    @Override
    /**
     * method: eval
     * usage: program.eval(env)
     * returns the value of the variable by getting it
     * from the given env
     * @param env   environment used for variable definitions
     * @return value of the variable from env
     */
    public int eval(Environment env)
    {
        return env.getVariable(name);
    }
}
```

Number.java

```java
package ast;

import environment.Environment;

/**
 * Number class that represents a basic number
 * in the form of an integer
 * when evaluated it just returns the integer value.
 * @author Jonathan Lee
 * @version 9 Oct2014
 *
 */
public class Number extends Expression
{
    private int value;

    /**
     * method: Number
     * constructor method that just
     * sets the value of the number
     * @param value value to be set
     * postcondition: value set
     */
    public Number(int value)
    {
        this.value = value;
    }

    @Override
    /**
     * method: eval
     * usage: program.eval();
     * evalues the number by returning its integer value
     * as the isntance field
     * @param env   environment used for variable definitions
     * postcondition: number evaluated by returning integer value
     */
    public int eval(Environment env)
    {
        return value;
    }
}
```

Scanner.java

```java
package scanner;

import java.io.*;

/**
 * ScannerLabAdvanced
 * Scanner is a simple scanner for Compilers and Interpreters (2014-
2015) lab exercise 1
 * @author Jonathan Lee
 *
 *  THIS ADVANCED SCANNER DOES NOT ACCOUNT FOR COMMENTED CODE
 *
 * <Insert a comment that shows how to use this object>
 * This object is used by making a scanner object from the class. one can then call next token to
 * retrieve tokens from the given source code. if any tokens are invalid, it will return a scan error exception
 * tokens are returned in string form
 * this object also ignores line and block comments
 *
 * LIST OF ALL REGULAR EXPRESSIONS USED (WILL BE UPDATED WITH EACH ITERATION):
 *  - letters := [a-z A-Z]
 *  - digits := [1-9]
 *  - operators := [=, +, -, *, /, %, (, ), {, }, <, >, ., &&, ||, >=, <=, ;, !, ==, !=, :=, -=, +=, //, /*, \*\/]
 *
 * PLEASE NOTE: THIS SCANNER CLASS WILL RECOGNIZE MULTIPLE CHARACTER OPERANDS WHICH WAS SPECIFIED IN THE LAB
 * DOCUMENT. THEREFORE, THIS CLASS TAKES A DIFFERENT, MORE COMPLICATED APPROACH TO RETURNING OPERAND TOKENS
 * THAT INVOLVES STRINGS NOT CHARACTERS. SEE THE scanOperand METHOD DOCUMENTATION FOR MORE INFORMATION REGARDING
 * THIS
 * A simpler version of the scanner was also made that only returns operands as single characters in the ScannerLab Project
 *
 *
 * @version Aug 29 2014 - file created and all basic methods implemented
 * @version Sep 2 2014 - added tokens to operator RegEx, updated documentation
 */
public class Scanner
{


    private BufferedReader in;
    private char currentChar;
    private boolean eof;
    /**
     * Scanner constructor for construction of a scanner that
     * uses an InputStream object for input.
     * Usage:
     * FileInputStream inStream = new FileInputStream(new File(<file name>));
     * Scanner lex = new Scanner(inStream);
     * @postcondition: scanner object created, currentChar defined
     * @param inStream the input stream to scan
     */
    public Scanner(InputStream inStream)
    {
        setInput(inStream);
        eof = false;
        getNextChar();
    }
    /**
     * Method: Scanner
     * Usage: new Scanner(string)
     * Scanner constructor for constructing a scanner that
     * scans a given input string.  It sets the end-of-file flag an then reads
     * the first character of the input string into the instance field currentChar.
     * Usage: Scanner lex = new Scanner(input_string);
     * @postcondition: scanner object created, currentChar defined
     * @param inString the string to scan
     */
    public Scanner(String inString)
    {
        setInput(inString);
        eof = false;
        getNextChar();
    }


    /**
     * method: setInput
     * usage: program.setInput(stream)
     * public helper method used to set the input
     * that the scanner reads to what is passed to the scanner
     * if the input is from a file, it will be an input stream
     * if it is from a string, it will be read as a string
     * input is set to the instance variable for use in the scanner
     * @param inStream  input stream from a file
     * postcondition: in set to new input stream
     */
    public void setInput(InputStream inStream)
    {
        in = new BufferedReader(new InputStreamReader(inStream));
    }

    /**
     * method: setInput
     * usage: program.setInput(string)
     * see setInput method description
     * @param inString inString  general input stream
     * postcondition: in set to the new input stream of the string
     */
    public void setInput(String inString)
    {
        in = new BufferedReader(new StringReader(inString));
    }

        /**
         * Method: getNextChar
```

```java
     * calls the read method from in (the instance variable that is a buffered reader)
     * in returns an int (if this int is -1, the reader is at the end of the file so eof can be set to true)
     * else, currentChar is set to the character value of the give read output
     * Method does not take any parameters or return an output
     * @precondition: has a next character
     * @postcondition: sets the currentchar to the next char in the source
     * @throws IOException in the event that the reader cannot obtain the next character for some reason
     */
    private void getNextChar()
    {
        int readChar = -1;      //base case, to avoid being marked as an error

        //try-catch-finally block intended to catch any error
        //that would occur from in.read()
        try
        {
            readChar = in.read();
        }
        catch (IOException e)
        {
            System.exit(0);
        }
        finally
        {
            //do nothing (program should crash if there is an io error)
        }


        if(readChar < 0)
            eof = true;
        currentChar = (char)readChar;
    }


    /**
     * Method: eat
     * moves the scanner along the file by calling getNextChar
     * however, before doing so, it will check to make sure its given char "expected"
     * matches the currentChar before moving on
     * @precondition: expected = currentChar
     * @postcondition: gets the next character in the source
     * @param expected - the expected character that the scanner should receive to match the current character (gives the green light to move on)
     * @throws ScanErrorException - thrown when expected != currentChar
     */
```

```java
        private void eat(char expected) throws ScanErrorException
        {
            if(expected == currentChar)
                getNextChar();
            else
                throw new ScanErrorException("Illegal character - expected " + currentChar + " and found " + expected + ".");
        }
        /**
         * Method: hasNext
         * used to determine whether the scanner has a next character to read
         * the opposite of eof is returned because eof keeps track of whether or not the reader
         * is at the end of the file (i.e. if eof is true, there is no next character so return false)
         * @return !eof - the opposite of the boolean that tracks if the reader is at the end of the file
         */
        public boolean hasNext()
        {
            return !eof;
        }

        /**
         * Method: isDigit
         * used to determine if the given character is a digit in the source

         * does this by checking it through the regular expression [0-9] (i.e. the digit is something between 0 and 9 inclusive)
         * returns true, if it is between (inclusive) those
         * returns false if it is not between (inclusive) those
         * RegEx
         * digits := [1-9]
         * @param c - the given character to test whether it is a digit or not
         * @return boolean that indicates whether it is a digit or not
         */
        public static boolean isDigit(char c)
        {
            return (c >= '0' && c <= '9');
        }

        /**
         * Method: isLetter
         * used to determine if the given character is a letter in the source
         * does this by checking it through the regular expression [a-z A-Z] (ie all the letters of the english alphabet
         * including upper case letters)
         * returns true if it is a letter (of either case)
         * returns false if it is not a letter
         * RegEx
```

```java
     * letters := [a-z A-Z]
     * @param c - the given character to test whether it is a letter or not
     * @return boolean that indicates whether it is a letter or not
     */
    public static boolean isLetter(char c)
    {
        return ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'));
    }

    /**
     * Method: isWhiteSpace
     * used to determine if the given character is a form of whitespace
     * does this by checking it through the regular expression [' ' '\r' '\n' '\t'] (i.e. all possible whitespace combinations)
     * returns true if it is a whitespace
     * returns false if it is not
     * RegEx
     * whitespace := [' ' '\r' '\n' '\t']
     * @param c - the given character to test whether it is whitespace or not
     * @return boolean that indicates whether it is a whitespace or not
     */
    public static boolean isWhiteSpace(char c)
    {
        return (c == ' ' || c == '\n' || c == '\r' || c == '\t');
    }

    /**
     * Method: nextToken
     * used to get the next token from the source. (propagates the scan exception that could occur)
     * does this by looping through whitespace until it hits the start of a potential number, identifier, operand, etc
     * once found, it will determine if it is the start of a specific token, and then scans it with one of the scan methods
     * returns the token
     * if the token happens to be an comment operand, it will use its helper handler methods
     * to ignore the appropriate source that is considered a comment (even accounts for nested comment blocks)
     * if end of file is reached, it returns "END"
     * @precondition: currentChar is defined
     * @postcondition: currentChar is progressed (to end of file or end of token), token returned
     * @return token - the token that was constructed based on the next character after whitespace
     * @throws exception - ScanErrorException thrown when unidentified token combination/syntax is found in scan methods
     */
    public String nextToken() throws ScanErrorException
    {
        String token = "";
        while(hasNext() && isWhiteSpace(currentChar))
        {
            eat(currentChar);
        }
        if(hasNext())
        {
            if(isDigit(currentChar))
                token = scanNumber();
            else if(isLetter(currentChar))
                token = scanIdentifier();
            else
            {
                token = scanOperand();
                /**if(token.equals("//"))
                {
                    handleLineComment();
                    token = nextToken();
                }
                else if(token.equals("/*"))
                {
                    handleBlockComment();
                    token = nextToken();
                }*/
            }
        }
        return token;
    }

    /**
     * Method: scanNumber
     * method called when the token is determined to be a number (ie starting with a digit)
     * this method will loop through the characters after it and appropriately append
     * them to a string that will result in the compiled number.
     * once it hits a token that would end the term (whitespace) the token is complete
     * and returned
     * if it runs into anything else (like a letter) in the middle of the number, it will throw an exception
     * number is only represented as (digit)(digit)*
     * invalid chars that can follow a number are: letters
     * @precondition: currentChar is not a letter or whitespace
     * @postcondition: currentChar is progressed, token returned
     * @return  token - the string comprised of the valid digits to make a number;
     * @throws ScanErrorException - thrown if the token is considered invalid (ie a token that starts with a number but has letters in it
     */
    public String scanNumber() throws ScanErrorException
    {
        String token = "" + currentChar;
```

```java
            eat(currentChar);
            while(hasNext() && isDigit(currentChar))
            {
                token += currentChar;
                eat(currentChar);
            }
            if(hasNext() && isLetter(currentChar))      //if a letter is placed in a  number, it is immediately invalid, this is not an acceptable token
                throw new ScanErrorException();
            else
                return token;
        }

        /**
         * Method: scanIdentifier
         * method called when token is determined to b a identifier (ie starting with a letter)
         * this method will loop through the characteers after it and appropriately append them
         * to a string that will result in a compiled identifier
         * once it hits a token that would end the term (whitespace) the token is compete and returned
         * invalid chars that can follow an identifier are:
         * @precondition: currentChar is not a digit or whitespace
         * @postcondition: currentChar is progressed, token returned
         * @return token - the string comprised of all the valid digits/letters to make an identifier;
         * @throws ScanErrorException - under conditions when following chars are not valid, this would be thrown (not as of now since any type of char can follow at the moment)
         */
        private String scanIdentifier() throws ScanErrorException
        {
            String token = "" + currentChar;
            eat(currentChar);
            while(hasNext() && isDigit(currentChar) || isLetter(currentChar))
            {
                token += currentChar;
                eat(currentChar);
            }
            //any other type of char can follow (including operands and whitespace, it just ends the token)
            return token;
        }

        /**
         * Method: scanOperand
         * method called when token is determined to be an operand (ie unidentified)
         * since operands are just single characters, no looping is needed as of now
         * since the operands can only be represented in an array, it searches throw this array for a matching operand
         * if none, given, it throws an exception
         * at the end, it must move to the next char so it eats
         * An exception will be thrown at any invalid character (i.e. not ones in the RegEx)
         * THIS METHOD WILL ONLY EAT (I.E. MOVE THE SCANNER ALONG) WHEN IT CONFIRMS THAT currentChar MAKES A VALID TOKEN.
         * Otherwise it will return the token without skipping characters.
         * By doing this, it will not overlook any characters if the operand fails at a certain length.
         *
         * RegEx
         * operators := [=, +, -, *, /, %, (, ), {, }, <, >, ., &&, ||, >=, <=, ;, !, ==, !=, :=, +=, -=, //, /*, \*\*]
         * @precondition currentChar is not a letter, digit or whitespace
         * @postcondition: currentChar is progressed, token returned
         * @return token - the string comprised of the operand
         * @throws ScanErrorException - under conditions when a character is not valid, exception is thrown
         */
        private String scanOperand() throws ScanErrorException
        {
            //char[] operand=  {'=', '+', '-', '*', '/', '%', '(', ')', ';', '!', ':'};             //array of valid operands
            //String[] compoundOperators = {"==", "===", ":=", "!=", "//", "/*", "*/"};
            String[] operators = {"=", "+", "-", "*", "/", "%", "(", ")", "{", "}", "<", ">", ".", "&&", "||", ">=", "<=", ";", "!", "==", "!=", ":=", "//", "/*", "*/", "+=", "-="};
            String token = "";
            int length = 1;
            String temp = "" + currentChar;

            int i = 0;
            while (i < operators.length
                    && hasNext()
                    && !isWhiteSpace(currentChar)
                    && !isLetter(currentChar)
                    && !isDigit(currentChar))
            {
                if((operators[i].length() >= length) && (operators[i].substring(0, length).equals(temp)))        //before testing, the operator's length has to be at least the length of currentChar, then test.
                {
                    if(length == operators[i].length())
                        token = temp;
                    eat(currentChar);
                    temp += currentChar;
                    length++;
                }
                else
                {
```

```java
                i++;        //don't increment if it is a match, because this term will have to be checked again
            }
        }

        if(token.length() > 0)
            return token;
        else
            throw new ScanErrorException();
    }


    /**
     * Method: handleLineComment
     * eats through all characters until it runs into a whitespace that is specifically "\n" because
     * that is the only character that will end the comment line, everything else is ignored
     * nothing is taken as a parameter and nothing is returned
     * @precondition: currentChar is immediately after the "//"
     * @postcondition: ate through everything on current line until "\n"
     */
    public void handleLineComment() throws ScanErrorException
    {
        while(currentChar != '\n' && hasNext())
            eat(currentChar);
    }

    /**
     * Method: handleLineComment
     * eats through all the characters until it runs into the next operator that would close
     * the comment which is the \*\/ character. everything within the block will be ignored.
     * @precondition: currentChar is immediately after a /* operand
     * @postcondition: ate through everything until close comment line
     */
    public void handleBlockComment() throws ScanErrorException
    {
        String operator = "";
        while (!operator.equals("*/") && hasNext())
        {
            if(!isWhiteSpace(currentChar)
                    && !isLetter(currentChar)
                    && !isDigit(currentChar))
            {
                operator = scanOperand();
            }
            eat(currentChar);
        }
    }


    }
```

ScanErrorException.java

```java
package scanner;

/**
 * ScanErrorException is a sub class of Exception and is thrown to indicate a
 * scanning error.  Usually, the scanning error is the result of an illegal
 * character in the input stream.  The error is also thrown when the expected
 * value of the character stream does not match the actual value.
 * @author Mr. Page
 * @version 062014
 *
 */
public class ScanErrorException extends Exception
{
    /**
     * default constructor for ScanErrorObjects
     */
    public ScanErrorException()
    {
        super();
    }
    /**
     * Constructor for ScanErrorObjects that includes a reason for the error
     * @param reason
     */
    public ScanErrorException(String reason)
    {
        super(reason);
    }
}
```

TEST CODE

INPUT

file.txt

```
display 3 read x
display x
assign a = 1
while a < x do
display a
assign a = a + 1
end
if x > 10 then
display x + (8-2) * 3 / 2
else
display -1
end
```

RESULTS

OUTPUT

```
file name: file.txt
3
20
20
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
29
```

The test code ran without errors and ran exactly the way the grammar for the SIMPLE interpreter specified as shown in the output figure. There were no thrown exceptions or errors and the program also successfully interprets variations of the input code.

CONCLUSIONS

Although the grammar posed several limitations for the interpreter, the modifications to the grammar detailed in the Grammar Transformations section solved these challenges. The foremost challenge was the left recursion that occurred as a byproduct of descending precedence. Again, simply altering the arrangement of the grammar terms resolved the issue.

Notably, the fact that the `Program` element has no definite end and that it is used not only for the program as a whole but also for other components of the grammar caused several debugging challenges. Firstly the `parseProgram` method has no way of determining the end of a program other than not receiving `Statement` input. However, this creates a conflict since sometimes the method will have to stop parsing `Statement` objects if the `Program` is embedded in an `If` statement or `While` statement despite not reaching the end of input. Therefore, there are keywords such as `else` and `end` that warrant neither the throwing of an exception nor the creation of a `Statement` object. Thus the interpreter's `parseStatement` method must recognize these certain keywords that essentially indicate no action and relay this indication to the `parseProgram` method, which then ceases parsing.

The limitations in the grammar pose few overall problems in the functionality of the interpreter. Each issue has a simple solution that does not alter the original grammar and functions as expected.