Project 2 Report

Blane Brown, Tucker Hoffnagle, Samantha Holmes

1. Introduction

This report covers the creation and testing of our group's recursive descent parser, implemented in java. The data structures in this project include the ones from our project 1 for the scan() function as well as a new one used to store the XML tree. Below are all data structures used:

1. nextStateTable

- nextStateTable is a 2d integer array with two indexes. The first dimension is indexed from 1 to 16 for the number of states. The second is indexed from 1 to 14 with 1 representing space/tab, 2 representing newline, 3 representing the character /, 4 representing the character *, 5 representing the character (, 6 representing the character), 7 representing the character +, 8 representing the character -, 9 representing the character :, 10 representing the character =, 11 representing the character ., 12 representing a digit, 13 representing any letter and 14 representing any other character.
- for any integer i and j, nextStateTable[i][j] is either a number greater than 0 meaning move to that state, a 0 meaning there is nowhere else to move and to recognize the token, or a -1 meaning the character is invalid (such as a '$').

2. tokenTypes

- tokenTypes is a one-dimensional string array containing every possible token type. The index of each token type correlates to the final state it represents.
-for any state i, tokenTypes[i] gives either the type of token that was found for that final state or an error token if the state given was not a final state.

3. Tree

- Tree is a StringBuffer that contains the formatted xml tree to represent our parser working.
- when a new recursive function call would occur, Tree would have appended to it the name of the function that was just called with proper indention, and once the call was complete that function would have its name appended once again to show that it has closed.

2. Pseudo-Code

2.1 main() method:

The main function is responsible for taking in an input file and printing either an XML tree if it follows correct syntax or “Error.” If it doesn’t.

main()

Input: a file F containing a program
Output: an XML tree showing how recursive descent parsing works, or error
Side Effect: the file F is read into a string S so that other functions can act on S.

Data:

S: input file characters written into a string (global string)
Input_token: current token read from S using scan() (global string)
tokenContent: content of the current token (global string)
Tree: XML tree to be printed (global stringbuffer)
counter: counts for size of indent (global int)
Result: result of calling Program() (either “ok” or “error”) (string)

Plan:

```
S := open file F and read its contents into S
Input_token := scan()
Result := Program()
If (Result not equal to “ok”)
        Print “Error.”
Else
        Print Tree
```

2.2 Program() method:

The Program function is responsible for seeing if an input file is accepted by the program or not using recursive descent parsing.

Program()

Input: acts on a global variable input_token
Output: either “ok” or “error” depending on if the program is accepted or not

Plan:

```
Tree.append(“<program>\n”)
counter++
switch (input_token)
        case id, read, write, $$: if (stmt_list() equals “ok”)
                                        Tree.append(“</program>\n”)
                                        Return match($$)
                                else
                                        Return “Error”
```

default case: return “Error”

2.3 stmt_list() method:

The stmt_list function is responsible for seeing if an input file is accepted by the program or not using recursive descent parsing.

stmt_list()

Input: acts on a global variable input_token

Output: either “ok” or “error” depending on if the program is accepted or not

Data:

answer: value returned from function call stored here so we can add to Tree the closing </method_name> before returning (string)

Plan:

```
Spacer(counter)
Tree.append(“<stmt_list>\n”)
counter++
switch (input_token)
        case id, read, write: if (stmt() equals “ok”)
                                        answer := stmt_list()
                             else
                                        answer := “Error”
        case $$: answer := “ok”
        default case: answer := “Error”
counter—
Spacer(counter)
Tree.append(“</stmt_list>\n”)
Return answer
```

2.4 stmt() method:

The stmt function is responsible for seeing if an input file is accepted by the program or not using recursive descent parsing.

stmt()

Input: acts on a global variable input_token

Output: either “ok” or “error” depending on if the program is accepted or not

Data:

answer: value returned from function call stored here so we can add to Tree the closing </method_name> before returning (string)

Plan:

```
Spacer(counter)
Tree.append(“<stmt>\n”)
```

```
counter++
switch (input_token)
        case id: match(id)
                if (match("assign") equals "ok")
                        answer := expr()
                else
                        answer := "Error"
        case read: match(read)
                answer := match(id)
        case write: match(write)
                answer := expr()
        default case: answer := "Error"
counter—
Spacer(counter)
Tree.append("</stmt >\n")
Return answer
```

2.5 expr() method:

The expr function is responsible for seeing if an input file is accepted by the program or not using recursive descent parsing.

expr()

Input: acts on a global variable input_token

Output: either "ok" or "error" depending on if the program is accepted or not

Data:

answer: value returned from function call stored here so we can add to Tree the closing </method_name> before returning (string)

Plan:

```
Spacer(counter)
Tree.append("<expr>\n")
counter++
switch (input_token)
        case lparen, id, number: if (term() equals "ok")
                                        answer := term_tail()
                                else
                                        answer := "Error"
        default case: answer := "Error"
counter—
Spacer(counter)
Tree.append("</expr >\n")
Return answer
```

2.6 term() method:

The term function is responsible for seeing if an input file is accepted by the program or not using recursive descent parsing.

term()

Input: acts on a global variable input_token

Output: either “ok” or “error” depending on if the program is accepted or not

Data:

answer: value returned from function call stored here so we can add to Tree the closing </method_name> before returning (string)

Plan:

```
Spacer(counter)
Tree.append(“<term>\n”)
counter++
switch (input_token)
        case lparen, id, number: if (factor() equals “ok”)
                                        answer := factor_tail()
                                else
                                        answer := “Error”
        default case: answer := “Error”
counter—
Spacer(counter)
Tree.append(“</term >\n”)
Return answer
```

2.7 term_tail() method:

The term_tail function is responsible for seeing if an input file is accepted by the program or not using recursive descent parsing.

term_tail()

Input: acts on a global variable input_token

Output: either “ok” or “error” depending on if the program is accepted or not

Data:

answer: value returned from function call stored here so we can add to Tree the closing </method_name> before returning (string)

Plan:

```
Spacer(counter)
Tree.append(“<term_tail>\n”)
counter++
switch (input_token)
```

```
        case minus, plus: if (add_op() equals “ok”)
                                if (term() equals “ok”)
                                        answer := term_tail()
                                else
                                        answer := “Error”
                        else
                                answer := “Error”
        case rparen, id, read, write, $$: answer := “ok”
        default case: answer := “Error”
    counter—
    Spacer(counter)
    Tree.append(“</term_tail >\n”)
    Return answer
```

2.8 factor() method:

The factor function is responsible for seeing if an input file is accepted by the program or not using recursive descent parsing.

factor()

Input: acts on a global variable input_token

Output: either “ok” or “error” depending on if the program is accepted or not

Data:

answer: value returned from function call stored here so we can add to Tree the closing </method_name> before returning (string)

Plan:

```
    Spacer(counter)
    Tree.append(“<factor>\n”)
    counter++
    switch (input_token)
        case lparen: match(lparen)
                        if (expr() equals “ok”)
                                answer := match(rparen)
                        else
                                answer := “Error”
        case id: answer := match(id)
        case number: answer := match(number)
        default case: answer := “Error”
    counter—
    Spacer(counter)
    Tree.append(“</factor >\n”)
    Return answer
```

2.9 factor_tail() method:

The factor_tail function is responsible for seeing if an input file is accepted by the program or not using recursive descent parsing.

factor_tail()

Input: acts on a global variable input_token

Output: either “ok” or “error” depending on if the program is accepted or not

Data:

answer: value returned from function call stored here so we can add to Tree the closing </method_name> before returning (string)

Plan:

```
Spacer(counter)
Tree.append(“<factor_tail>\n”)
counter++
switch (input_token)
        case times, div: if (mult_op() equals “ok”)
                                if (factor() equals “ok”)
                                        answer := factor_tail()
                                else
                                        answer := “Error”
                        else
                                answer := “Error”
        case plus, minus, rparen, id, read, write, $$: answer := “ok”
        default case: answer := “Error”
counter—
Spacer(counter)
Tree.append(“</factor_tail >\n”)
Return answer
```

2.10 add_op() method:

The add_op function is responsible for seeing if an input file is accepted by the program or not using recursive descent parsing.

add_op()

Input: acts on a global variable input_token

Output: either “ok” or “error” depending on if the program is accepted or not

Data:

answer: value returned from function call stored here so we can add to Tree the closing </method_name> before returning (string)

Plan:

```
Spacer(counter)
```

Tree.append(“<add_op>\n”)
counter++
switch (input_token)
case plus: answer := match(plus)
case minus: answer := match(minus)
default case: answer := “Error”
counter—
Spacer(counter)
Tree.append(“</add_op>\n”)
Return answer

2.11 mult_op() method:

The mult_op function is responsible for seeing if an input file is accepted by the program or not using recursive descent parsing.

mult_op()

Input: acts on a global variable input_token
Output: either “ok” or “error” depending on if the program is accepted or not

Data:

answer: value returned from function call stored here so we can add to Tree the closing </method_name> before returning (string)

Plan:

Spacer(counter)
Tree.append(“<mult_op>\n”)
counter++
switch (input_token)
case times: answer := match(times)
case div: answer := match(div)
default case: answer := “Error”
counter—
Spacer(counter)
Tree.append(“</mult_op >\n”)
Return answer

2.12 match() method:

The match function is responsible for seeing if the expected token is matched with the current input token

match()

Input: acts on a global variable input_token and an argument of expectedToken
Output: either “ok” or “error” depending on if the tokens match or not
Side Effect: Input token is updated to the next token using scan()

Plan:

```
If (expectedToken equals input_token)
        If (expectedToken not equal to $$)
                Spacer(counter)
                Tree.append("<expectedToken>\n")
                Spacer(counter)
                Tree.append("    tokenContent\n")
                Spacer(counter)
                Tree.append("</expectedToken>\n")
        Input_token := scan()
        Return "ok"
else
        Return "Error"
```

2.13 Spacer() method:

The spacer function is responsible for adding proper indentions into the XML tree

spacer()

Input: acts on a global variable counter
Output: none
Side Effect: indents are added to Tree

Plan:

```
For (int x := 0; x < counter; x++)
        Tree.append("    ")
```

2.14+ The Remaining Project 1 Methods:

I don't want to include them in this project report since the pseudo code section is already long enough, but scan() and its supporting methods from project 1 were used to create our recursive descent parser.

3. Test Cases

1. **read A**
   - The reason I selected this test case was because it is the example given in the project description, and since the project description shows how the output of this exact input should look, we can use it to test if our tree is printing properly.

2. **Sum := (8 + const) / 3**
   - The reason I selected this test case was because it is a more complex input that involves parenthesis and different types of operators. Testing this more complex program allows us to see what the output looks like to make sure all the different methods are working properly to print the correct tree.

3. **read read A**
   - The reason I selected this test case was because it is an example of an input program that should not work and should return error since read is being followed by another read and an ID and not just an ID.

4. **Sum := (3 * 4**
   - The reason I selected this test case was because it is an example of a input program that should fail due to one parenthesis missing, even though the rest of the syntax is ok. This is important to test because matching parenthesis is very important and this will show that our project is working properly.

5. **Empty input file**
   - The reason for testing an empty input file is that based on the given grammar in the project description parsing an empty file should not return error, so this can show that our program works properly with empty files.

4. Acknowledgement

Blane Brown: Implemented the recursive descent parser in Java and created the pseudo code for the project and report.

Tucker Hoffnagle: Implemented the XML tree printing into the recursive descent parser and helped fix bugs in the program.

Samantha Holmes: Created the project report and helped in the creation of the recursive descent parser