Project 2 Build a Parser

IMPLEMENTATION

Recursive Descent: We achieved a recursive descent approach by having functions that called other functions going down to the token level, and then if the token returned true, it called back up with trues and ran the code to have the first function return true. If one function returned false and there was no alternative rule to return true, the code would not run since the descent returned a false.

Parser Function: We interpreted the keyword 'parser' as the identifier for our function so all of our code is the function and typing the keyword 'parser' in the command line calls on our code.

Scan Function: We reused the scan function from Project 1.

PSEUDOCODE

Data (elements for scan function not given since they were on Project 1)

Given data:

file: the name of the file to create the parse tree for

Unknown data:

Intermediate data:

output: the array of XML formatted strings to output

fpIndex: the pointer to the character the file is on

whitespace: the array containing all whitespace strings

Plan

```
main(){
    Output = []
    Check for valid parser command and text file
    if(Program())
        Int counter = 0
        For item in output
            if(item[0:2]  != "</")
                For indent in range 0,counter
                    print(\t)
                print(item)
            Else
                counter-=1
                For indent in range 0,counter
                    print(\t)
                print(item)
    else
        Error
}

bool Program()
{
    output.append("<Program>")
```

```
        if(stmt_list())
                if($$)
                        output.append(“</Program>”)
                        Return true
                Else
                        Return false
        Else
                return false
}

Bool stmt_list()
{
        output.append(“<stmt_list>”)
        if(stmt())
                if(stmt_list())
                        output.append(“</stmt_list>”)
                        Return true
                Else
                        Return false
        elif(empty())
                output.append(“</stmt_list>”)
                Return True
        Else
                Return false
}

Bool stmt()
{
        output.append(“<stmt>”)
        if(scan() == “id” && scan() == “assign”)
                output.append(“id”)
                output.append(“assign”)
                If(expr())
                        output.append(“</stmt>”)
                        Return true
                else
                        Return false
        elif(scan() == “read” and scan() == “id”)
                output.append(“read”)
                output.append(“id”)
                output.append(“</stmt>”)
                Return true
        elif(scan() == “write” )
                output.append(“write”)
                If(expr())
                        output.append(“</stmt>”)
                        Return true
```

```
                Else
                        Return false
        Else
                Return false
}

Bool expr()
{
        output.append(“<expr>”)
        if(term())
                if(term_tail())
                        output.append(“</expr>”)
                        Return true
                Else
                        Return false

        elif(empty())
                output.append(“</expr>”)
                Return true
        Else
                Return false
}

Bool term_tail()
{
        output.append(“<term_tail>”)
        if(add_op())
                if(term)
                        if(term_tail())
                                output.append(“</term_tail>”)
                                Return True
                        Else
                                Return false
        elif(empty())
                output.append(“</term_tail>”)
                Return true
        Else
                Return false
}

Bool term ()
{
        output.append(“<term>”)
        if(factor())
                if(fact_tail())
                        output.append(“</term>”)
                        Return true
```

```
                Else
                        Return false
        Else
                Return false
}

Bool fact_tail()
{
        output.append(“<fact_tail>”)
        if(mult_op())
                if(factor())
                        if(fact_tail())
                                output.append(“</fact_tail>”)
                                Return true
                        Else
                                Return false
                Else
                        Return false
        elif(empty())
                output.append(“</fact_tail>”)
                Return true
        Else
                Return false
}

Bool factor()
{
        output.append(“<factor>”)
        if(scan() == “lparen”)
                if(expr())
                        if(scan() == “rparen”)
                                output.append(“</factor>”)
                                Return true
                        Else
                                Return false
                Else
                        Return false
        elif(scan() == “id”)
                output.append(“</factor>”)
                Return true
        elif(scan() == “number”)
                output.append(“</factor>”)
                Return true
        Else
                Return false
}
```

```
Bool add_op()
{
        output.append("<add_op>")
        if(scan() == "plus" || scan == "minus")
                output.append("</add_op>")
                Return true
        Else
                Return false
}

Bool mult_op()
{
        output.append("<mult_op>")
        if(scan() == "times" || scan() == "div")
                output.append("</mult_op>")
                Return true
        Else
                Return false
}

Bool empty()
{
        If whitespace
                Return true
        Else
                Return false
}
```

TEST CASES

The first test case was to incorrectly spell the word 'parser' to make sure that the first word being inputted to the console was parser as specified. Next we tested our input file to make sure that if an invalid file was typed in it prompts the user to enter a valid file and try again. Once we made sure that was working we began testing of the parser. First test of the parser was an id (x) only on the input file and we did this to make sure the parser correctly ran through and read the id and went from program to stmt_list then to stmt the printed the id(x) then closed the stmt then stmt_list the program. Once this was printing out the XML tree in correct format we proceeded with our test cases. Then we tested just 'read' to make sure that we get an error in parser. Next we tested "read A"and "read 5' to amke sure the proper xml tree was being displayed. Then we moved on to 'write' to make sure we got an error in parser. Next we tested 'write a' and 'write 5' to make sure the correct xml tree was displayed. Finally we tested assign and the operators with a combination of parenthesis, addition, subtraction, times, and divide. We ran countless tests with all sorts of combinations to make sure the correct xml tree was printed and also to make sure we got an error in parser if something wasn't correct.

ACKNOWLEDGEMENT
Jacob Strickland and Colin Morrison