Michael DeGraw
R1083006

Introduction

I found it to be the most simple to represent the machine as a series of recursive function calls. This is because there is an implied stack. For instance, in evaluating each state, it is important to consider the current state of the stack, and determine where to go from there. Because each function IS the state, all that is needed is where to go. For instance, in the case of a <stmt>, a <stmt> must be one of three things. There must be an id currently in the input stream, there must be a read currently in the input stream, or there must be a write in the input stream. In the event that none of these three events occur, and we are evaluating a <stmt> then there is an error.

Pseduo-code

scan // assumed functionality from Project 1
Input: a string of tokens stored in array s[]
Output: a single token if no error, otherwise “error”
end of scan

buildParseTree // the main function/algorithm
Input: a text file
Output: a parse tree generated from the text file displayed on console
Data: inputString: a multi-line string containing the program
Plan:
// all programs begin the same
print(program(inputString));
end of buildParseTree

program
Input: a string containing the entire input file
Output: a string containing either our parse tree or “error”
Data: inputString: the string we read from the file
outputString: the string containing the output
Plan:
outputString = “<program>\n”;
stmtList(outputString, inputString, 1);
// check to see if we have been returned an error
if (outputString == "error"){
return “error”;
}
outputString += “</program>\n”;
return outputString;
end of program

stmtList
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output

indentLevel: the current tier of indentation

Plan:

```
outputString = format(“stmt_list”, indentLevel);
indentString(outputString, indentLevel);
if (scan(inputString) != NULL){
        outputString = stmt(inputString, outputString, indentLevel + 1);
        outputString = stmt_list(inputString, outputString, indentLevel + 1);
}
// check to see if we have been returned an error
if (outputString == "error"){
        return “error”;
}
outputString = format(“/stmt_list”, indentLevel);
return outputString;
```

end of stmtList

stmt

Input: an input string, an output string, an indent level

Output: a modified output string

Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation

Plan:

```
outputString = format(“stmt”, indentLevel);
if (scan(inputString) == id){
        outputString = id(inputString, outputString, indentLevel +1 );
        outputString = assign(inputString, outputString, indentLevel +1 );
        outputString = expr(inputString, outputString, indentLevel + 1);
} else if (scan(inputString) == read){
        outputString = read(inputString, outputString, indentLevel + 1);
        outputString = id(inputString, outputString, indentLevel + 1);
} else if (scan(inputString) == write){
        outputString = write(inputString, outputString, indentLevel + 1);
        outputString = expr(inputString, outputString, indentLevel + 1);
} else {
        return “error”;
}
// check to see if we have been returned an error
if (outputString == "error"){
        return “error”;
}
outputString += format(“/stmt”, indentLevel);
return outputString;
```

end of stmt

expr

Input: an input string, an output string, an indent level

Output: a modified output string

Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
outputString += format(“expr”, indentLevel);
outputString = term(inputString, outputString, indentLevel + 1);
// check to see if we have been returned an error
if (outputString == "error"){
        return “error”;
}
outputString = term_tail(inputString, outputString, indentLevel +1);
// check to see if we have been returned an error
if (outputString == "error"){
        return “error”;
}
outputString += format(“/expr”, indentLevel);
return outputString;
```

end of expr

term_tail
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
outputString += format(“term_tail”, indentLevel);
if (scan(inputString) != NULL){
        outputString = add_op(inputString, outputString, indentLevel + 1);
        // check to see if we have been returned an error
        if (outputString == "error"){
                return “error”;
        }
        outputString = term(inputString, outputString, indentLevel + 1);
        // check to see if we have been returned an error
        if (outputString == "error"){
                return “error”;
        }
        outputString = term_tail(inputString, outputString, indentLevel+1);
        // check to see if we have been returned an error
        if (outputString == "error"){
                return “error”;
        }
}
outputString += format(“/term_tail”, indentLevel);
return outputString;
```

end of term_tail

term
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
outputString += format(“term”, indentLevel);
outputString = factor(inputString, outputString, indentLevel + 1);
// check to see if we have been returned an error
if (outputString == "error"){
        return “error”;
}
outputString = fact_tail(inputString, outputString, indentLevel + 1);
// check to see if we have been returned an error
if (outputString == "error"){
        return “error”;
}
outputString += format(“/term”, indentLevel);
return outputString;
```

end of term

fact_tail
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
outputString += format(“fact_tail”, indentLevel);
if (scan(inputString) != NULL){
        outputString = mult_op(inputString, outputString, indentLevel + 1);
        // check to see if we have been returned an error
        if (outputString == "error"){
                return “error”;
        }
        outputString = factor(inputString, outputString, indentLevel + 1);
        // check to see if we have been returned an error
        if (outputString == "error"){
                return “error”;
        }
        outputString = fact_tail(inputString, outputString, indentLevel + 1);
        // check to see if we have been returned an error
        if (outputString == "error"){
                return “error”;
        }
}
outputString += format(“/fact_tail”, indentLevel);
```

return outputString;
end of fact_tail

factor
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
outputString += format(“factor”, indentLevel);
if (scan(inputString) == lparen){
        outputString = lparen(inputString, outputString, indentLevel + 1);
        // check to see if we have been returned an error
        if (outputString == "error"){
                return “error”;
        }
        outputString = expr(inputString, outputString, indentLevel + 1);
        // check to see if we have been returned an error
        if (outputString == "error"){
                return “error”;
        }
        outputString = rparen(inputString, outputString, indentLevel + 1);
} else if (scan(inputString) == id){
        outputString = id(inputString, outputString, indentLevel + 1);
} else if (scan(inputString) == number){
        outputString = number(inputString, outputString, indentLevel + 1);
} else {
        return “error”;
}
// check to see if we have been returned an error
if (outputString == "error"){
        return “error”;
}
outputString += format(“/factor”, indentLevel);
return outputString;
```

end of factor

add_op
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
outputString += format(“add_op”, indentLevel);
if (scan(inputString) == plus){
        outputString = plus(inputString, outputString, indentLevel + 1);
```

```
} else if (scan(inputString) == minus){
        outputString = minus(inputString, outputString, indentLevel + 1);
} else {
        return “error”;
}
// check to see if we have been returned an error
if (outputString == "error"){
        return “error”;
}
outputString += format(“/add_op”, indentLevel);
return outputString;
```
end of add_op

mult_op
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
outputString += format(“mult_op”, indentLevel);
if (scan(inputString) == times){
        outputString = times(inputString, outputString, indentLevel + 1);
} else if (scan(inputString) == div){
        outputString = div(inputString, outputString, indentLevel + 1);
} else {
        return “error”;
}
// check to see if we have been returned an error
if (outputString == "error"){
        return “error”;
}
outputString += format(“/mult_op”, indentLevel);
return outputString;
```
end of mult_op

```
//
// now we get to terminals
//
```

id
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
if (scan(inputString) == id){
```

```
        outputString += format(“id”, indentLevel);
        outputString += indentString(indentLevel + 1);
        outputString += id.value; // the value of the token i.e. 1 if this were number
        outputString += format(“/id”, indentLevel);
        remove value from inputString;
        return outputString;
} else {
        return “error”;
}
```
end of id

assign
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
if (scan(inputString) == assign){
        outputString += format(“assign”, indentLevel);
        outputString += indentString(indentLevel + 1);
        outputString += “=”;
        outputString += format(“/assign”, indentLevel);
        remove value from inputString;
        return outputString;
} else {
        return “error”;
}
```
end of assign

read
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
if (scan(inputString) == read){
        outputString += format(“read”, indentLevel);
        outputString += indentString(indentLevel + 1);
        outputString += “read”;
        outputString += format(“/read”, indentLevel);
        remove value from inputString;
        return outputString;
} else {
        return “error”;
}
```
end of read

write
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
if (scan(inputString) == write){
        outputString += format("write", indentLevel);
        outputString += indentString(indentLevel + 1);
        outputString += "write";
        outputString += format("/write", indentLevel);
        remove value from inputString;
        return outputString;
} else {
        return "error";
}
```

end of write

lparen
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
if (scan(inputString) == lparen){
        outputString += format("lparen", indentLevel);
        outputString += indentString(indentLevel + 1);
        outputString += "(";
        outputString += format("/lparen", indentLevel);
        remove value from inputString;
        return outputString;
} else {
        return "error";
}
```

end of lparen

rparen
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
if (scan(inputString) == rparen){
        outputString += format("rparen", indentLevel);
```

```
        outputString += indentString(indentLevel + 1);
        outputString += “)”;
        outputString += format(“/rparen”, indentLevel);
        remove value from inputString;
        return outputString;
} else {
        return “error”;
}
```
end of rparen

number
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
if (scan(inputString) == number){
        outputString += format(“number”, indentLevel);
        outputString += indentString(indentLevel + 1);
        outputString += number.value;
        outputString += format(“/number”, indentLevel);
        remove value from inputString;
        return outputString;
} else {
        return “error”;
}
```
end of number

plus
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
if (scan(inputString) == plus){
        outputString += format(“plus”, indentLevel);
        outputString += indentString(indentLevel + 1);
        outputString += “+”;
        outputString += format(“/+”, indentLevel);
        remove value from inputString;
        return outputString;
} else {
        return “error”;
}
```
end of plus

minus
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
if (scan(inputString) == minus){
        outputString += format(“minus”, indentLevel);
        outputString += indentString(indentLevel + 1);
        outputString += “-”;
        outputString += format(“/minus”, indentLevel);
        remove value from inputString;
        return outputString;
} else {
        return “error”;
}
```

end of minus

times
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
if (scan(inputString) == times){
        outputString += format(“times”, indentLevel);
        outputString += indentString(indentLevel + 1);
        outputString += “*”;
        outputString += format(“/times”, indentLevel);
        remove value from inputString;
        return outputString;
} else {
        return “error”;
}
```

end of times

div
Input: an input string, an output string, an indent level
Output: a modified output string
Data: inputString: the string we read from the file
outputString: the string containing the output
indentLevel: the current tier of indentation
Plan:

```
if (scan(inputString) == div){
        outputString += format(“div”, indentLevel);
        outputString += indentString(indentLevel + 1);
```

```
        outputString += "/";
        outputString += format("/div", indentLevel);
        remove value from inputString;
        return outputString;
} else {
        return "error";
}
```
end of div

format
Input: a string value, an indent level
Output: a modified string
Data: str: what is going in angle brackets
indentLevel: the number of indents to use
Plan:
outputString += indentString(indentLevel);
outputString += "<" + str + ">";
end of format

indentString // function used to indent a string
Input: string, number of indents
Output: the same string variable now indented
Plan:
for (int x = 0; x < indent; x++){
        string += " ";
}
return string;
end of indentString

Test Cases

A := 5 *first case of <stmt>*
read A *second case of <stmt>*
write 5 *third case of <stmt>*
A := (5) *first case of <factor>*
A := X *second case of <factor>*
A := 5 *third case of <factor>*
A := 5+2 *first case of <add_op>*
A := 5-2 *second case of <add_op>*
A := 5*2 *first case of <mult_op>*
A := 5/2 *second case of <mult_op>*
write (X+5)/(2+5*10) read B *complex test case*

Acknowledgement

I was the only one who contributed to this project