INTERPRETER/COMPILER FOR MINI LANGUAGE

Melissa Rodriguez, Elikem Kuiv, Gregory Parker

Table of Contents

Overall Structure

Scanner

Takes in a sample program in the form of a text file and reads in the text file character by character. As the scanner reads the text file character by character, it determines all the integer value (token) of each character in the sample program and outputs the tokens to a file ("filename-output.txt"). Note that the tokens outputted to "filename-output.txt" are the integer codes of the characters. The glossary for the token codes can be found in section 4 of this document.

The scanner is implemented in the Interpreter class in the Scanner() method and uses the charBufferedReader class to read in the text file character by character. The scanner consists of a series of if-else statements to determine token for each character(s) read in, and at the same time checks for characters found in the sample program that are not part of the project language grammar.

Parser

The parser takes in the file of tokens created by the scanner ("filename-output.txt") and reads the tokens one by one. As the parser reads in tokens it checks to see if the program is syntactically correct according to the language grammar.

The parser is implemented in several methods in the Interpreter class. There is roughly one method for every grammar rule, which determines if the order of the tokens found is a valid part of the grammar. The parser is started when program() in main(String args[]) is called. Program() calls the appropriate grammar rule methods as it reads in tokens. If one of the methods fails the program will output "PARSER: FAIL"; however, if all the methods called by program() are true then the program will output "PARSER: SUCCESS".

Postfix

The postfix part of the program arranges the tokens found by the scanner in the order in which they should be executed, and puts them in an array called postfix[]. For example, an operation token is immediately executed as soon as it is found in the postfix array.

The postfix generation is implemented within the parser methods for given grammar rules. In the postfix array, identifiers, integer constants, word elements (letter & digits within quotes), and set elements are stored as two part tokens, while all operations are stored as one-part tokens. For example, for the identifier x to be added to the postfix array, the the first part of the token should be the x's type, and the second part should be x's position in the symbol table. For a constant to be added, the the first part of the token should be the token code for constant, and the second token should be the actual value of the constant.

Branching tokens were created for this part of the program to be able to handle how loops and conditional statements should be executed. The glossary for the postfix token codes can be found in section 4 of this document.

Symbol Table

The Symbol Table holds the identifiers declared in the sample program. The symbol table is implemented in the SymbolTable class, and here an array of SymbolTableElement objects is created to hold the identifiers.

The SymbolTableElement class represents an identifier object. Some of the attributes associated with the SymbolTableElements are type, name, value, and, max_size and curr_size (for set identifiers).

The identifiers are added to the symbol table in the Parser part of the program, and the values of the identifiers are modified in the execution part of the program whenever an assign operation occurs.

Executing test programs

The last part of the program is to execute the sample program text file that was read in by the scanner.

The implementation for executing the sample program text file is done in the Execution class. A 2D array called runStack[][] is defined in this class. RunStack[][] holds the operands and jump positions so that when an operation token is discovered, the elements at the top of the runStack are computed with the discovered operation.

The method RunProgram() takes in the postfix array and reads in each token. RunProgram() consists of a switch statement which evaluates the token read in, so as to decide which switch statement case the currently read token falls into. Each case takes care of adding the appropriate elements to the runStack and computing the appropriate operations. The symbol table is also updated when the sample program assigns a value to an identifier.

Limitations

Features from the project language grammar that are not implemented in the program are:

All the operations for integers can be executed except

- read
- random
- Binary operations for sets can only be done between identifiers declared as sets, or between sets not declared as identifiers. Binary operations between an identifier declared as a set, and a set not declared as an identifier cannot be executed. For example:
    - ❖ The compiler can execute "declare set 10 a, b, c $ a := {1,2,3,}; b := {4,5,6}; c := a +b ##"
    - ❖ The compiler can execute "declare set 10 a $ a := {1,2,3} + {4,5,6} ##"
    - ❖ However, the compiler cannot execute "declare set 10 a, b $ a := {1,2,3,4}; b := a + {5,6,7} ##"
- Conditional statements for sets does not work.

Program Execution Instructions and Program Output

Instructions

Upon running the program the user will be prompted to enter a text file to be read in – this should be a text file of a sample program. This text file is then executed by this compiler. There are sample programs included in section 5 of this document that can be used for testing. After, the program will create a file whose name will take the format, "inputtedtextfilename-output.txt". This file is where the tokens read in will be written to. This is also the same file the parser will take in to do error-checking of the sample program'. If the parser is successful, the execution part of the program will take in the postfix array generated and compute all the operations in the sample program.

Output

Besides having the token codes outputted to a text file, the token codes will also be printed to the IDE's console. Right below the token codes, the actual tokens (String version) will be printed so as to annotate the token codes. Next, a message will be printed by the parser to notify the user as to whether it was successful or not. A failure in the parser implies that the sample program contains syntax errors. If the parser notifies a failure, then the program will not be executed. However, if the parser notifies that it was successful, then the postfix array will be printed next. Finally the output of the actual sample program will be printed. There will only be output from the sample program if there is a write operation in the sample program. In case there is no write statement, the symbol table will also be printed to show the values of the identifiers after all the operations in the sample program were executed.

Token Code Glossary

Symbol Table Token Code Glossary

Symbols	Token Int
EOF	0
IDENTIFIER	1
Constant	2
SET ELEMENT	50
WORD ELEMENT	128
DOLLAR	99
ERROR	100

Symbols	Token Int	Language
ASSIGN	3	:=
ENDPROG	4	##
GREATEREQ	5	>+
GREATER	6	>
LESSEQ	7	<=
LESS	8	<
CONT	9	..
COMMENT	10	–
MINUS	11	-
PLUS	12	+
MULT	13	*
DIV	14	/
LPAR	15	(
RPAR	16	)
LBAC	17	{
RBRAC	18	}
EQUAL	19	=
COMMA	20	,
SEMI	21	;
QOUTE	22	'
NOT	48	#

Keywords	Token Int	Language
KW_IF	23	IF
KW_ELSE	24	ELSE
KW_READ	25	READ
KW_WRITE	26	WRITE
KW_DECLARE	27	DECLARE
KW_RANDOM	28	RANDOM
KW_FOR	29	FOR
KW_TO	30	TO
KW_THEN	31	THEN
KW_LOOP	32	LOOP
KW_FI	33	FI
KW_ENDLOOP	34	ENDLOOP
KW_SET	35	SET

Parser Table Token Code Glossary

Postfix Keywords	Token Int
BR	38
BMZ	39
BM	40
BPZ	41
BP	42
BZ	43
BNZ	44
UMINUS	45
READ	46
WRITE	47

Sample Programs and Output Generated

(To execute sample code, type them in a text editor like notepad; do not copy and paste from a word processor because word processors further format characters and this ends up making characters recognizable by the compiler)

test.txt

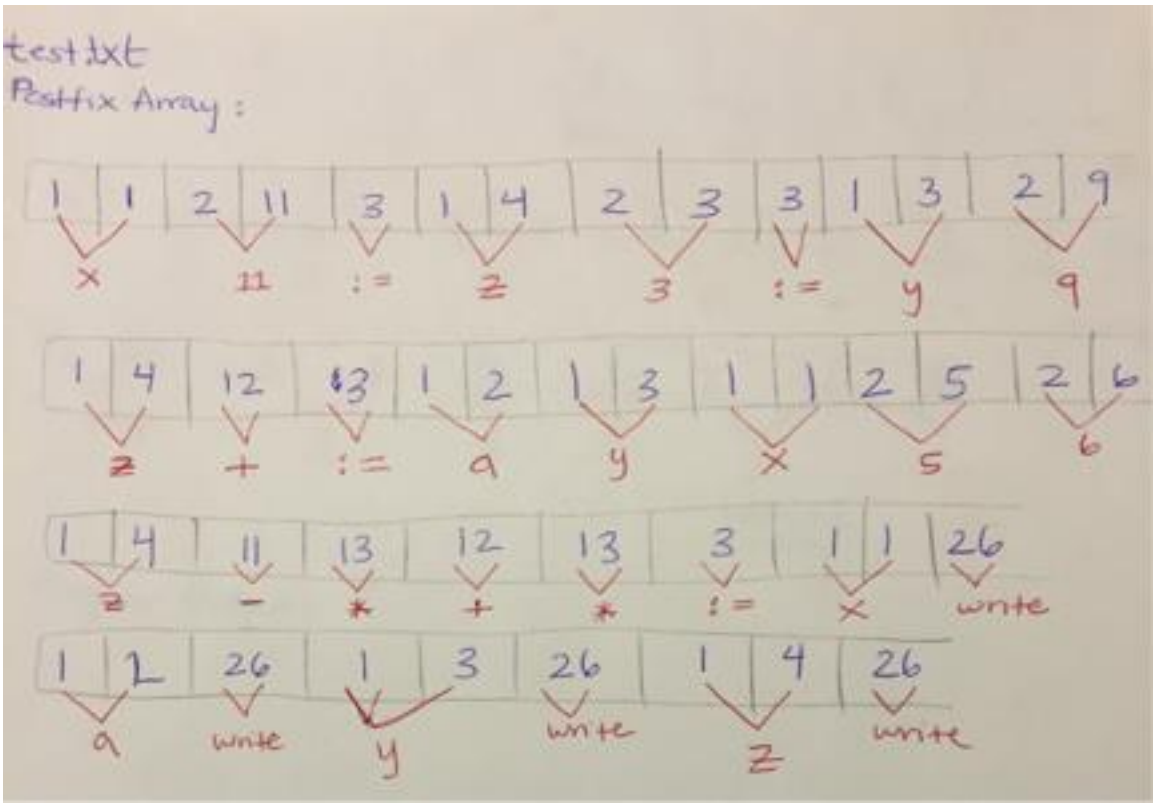
```
declare integer x, a, y, z

$

x:= 11;
z:= 3;
y:= 9+z;
a:= y * (x+5*(6-z));

write x, a, y, z

##
```



Program Output:

11
312
12
3

Symbol Table:

type: 36, name: x, value: 11
type: 36, name: a, value: 312
type: 36, name: y, value: 12
type: 36, name: z, value: 3

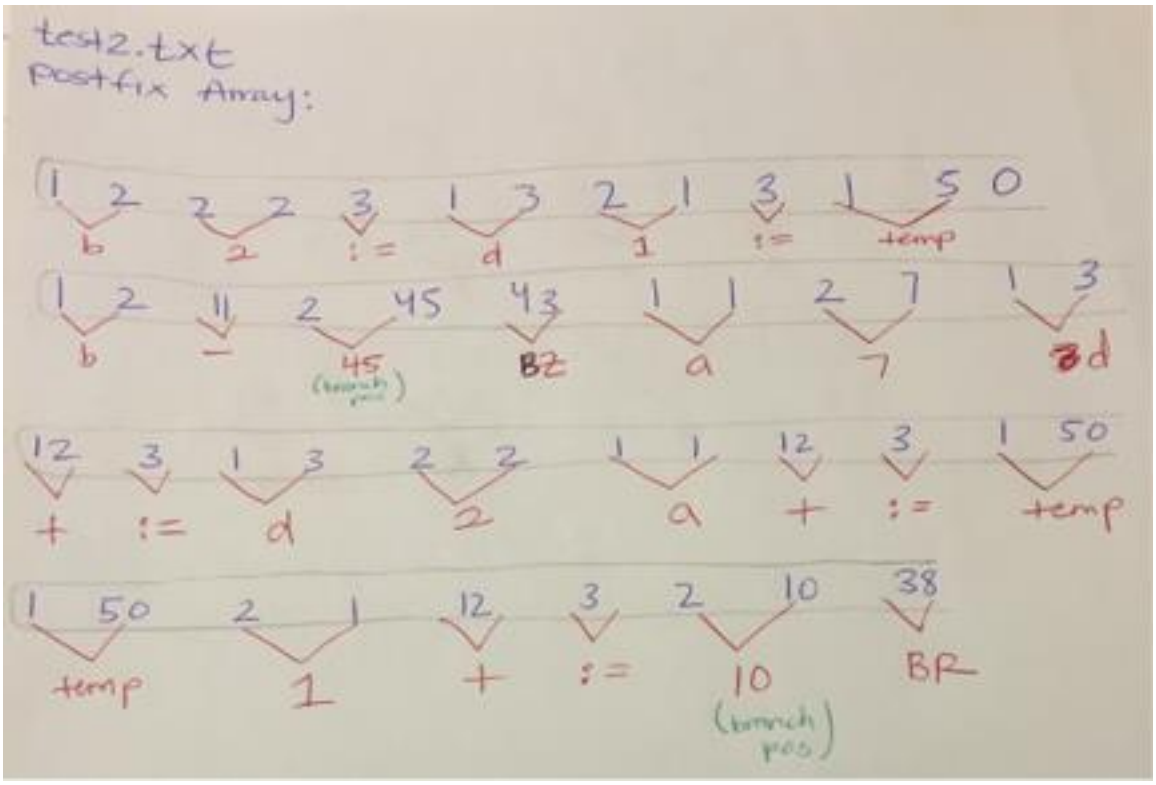
test2.txt

```
declare integer a, b, d

$

b:= 2;
d:= 1;
to b loop
        a:= 7+d;
        d:= 2+a
endloop

##
```



Program Output:

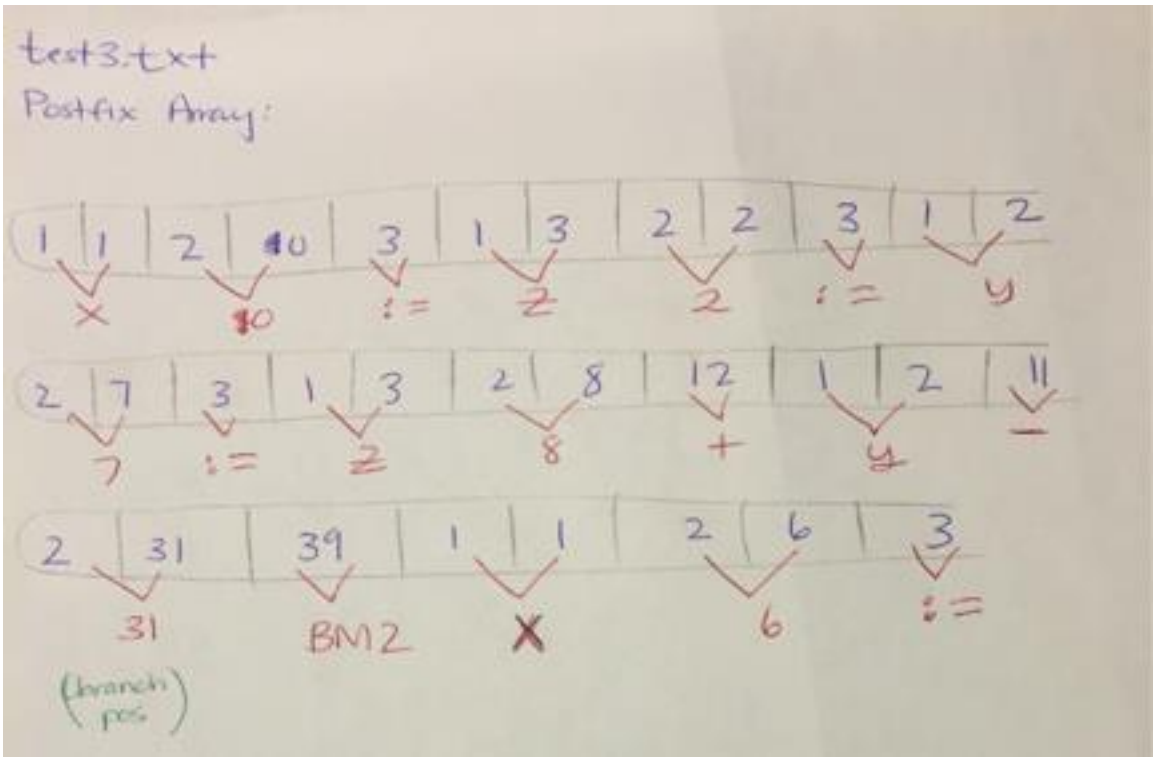
Symbol Table:

type: 36, name: a, value: 17
type: 36, name: b, value: 2
type: 36, name: d, value: 19

test3.txt

```
declare integer x, y, z
$
x:= 0;
z:= 2;
y:= 7;
if z+8 > y then
        x:= 6
fi

##
```



Program Output:

Symbol Table:

type: 36, name: x, value: 6
type: 36, name: y, value: 7
type: 36, name: z, value: 2

test4.txt

```
declare integer x, y, z
$
x:= 0;
z:= 2;
y:= 7;
if z > y then
        x:= 6
else
        x:= 100;
        y:= x/z
fi

##
```

Program Output:

Symbol Table:
type: 36, name: x, value: 100
type: 36, name: y, value: 50
type: 36, name: z, value: 2

test5.txt

```
declare set 10 x,y

$

x := {1,2,3,4,5,6,7};
y := {1,3,4,5,6,7,9}

##
```

Program Output:

Symbol Table:
type: 35, name: x, max_size: 10, current_size: 5, value: {5,4,3,2,1}
type: 35, name: y, max_size: 10, current_size: 4, value: {10,8,9,10}

test6.txt

```
declare set 10 x

$

x := {1,2,3,4,5,6,7} + {1,7,5,10,11,15,20}

##
```

Program Output:

Symbol Table:
type: 35, name: x, max_size: 10, current_size: 9, value: {1,2,4,5,7,10,11,15,20}

test7.txt

```
declare set 10 x

$

x := {1,2,3,4,5,6,7} * {1,7,5,10,11,15,20}

##
```

Program Output:

Symbol Table:

```
type: 35, name: x, max_size: 10, current_size: 3, value: {1,7,5}
```

test8.txt

```
declare set 10 a, b, c

$

a := {1,2,3,4,5};
b := {6,2,3,7,8};
c := a + b

##
```

Program Output:

Symbol Table:

```
type: 35, name: a, max_size: 10, current_size: 5,value: {1,2,3,4,5}
type: 35, name: b, max_size: 10, current_size: 5,value: {6,2,3,7,8}
type: 35, name: c, max_size: 10, current_size: 8,value: {1,2,3,4,5,6,7,8}
```

test9.txt

```
declare set 10 a, b, c

$

a := {1,2,3,4,5};
b := {6,2,3,7,8};
c := a * b

##
```

Program Output:

Symbol Table:

```
type: 35, name: a, max_size: 10, current_size: 5,value: {1,2,3,4,5}
type: 35, name: b, max_size: 10, current_size: 5,value: {6,2,3,7,8}
type: 35, name: c, max_size: 10, current_size: 2,value: {2,3}
```