Retro Basic Compiler Project

1. A scanner

In the source (Retro Basic), instructions will be separated with space (“ “) or new line (“\n”). So, to separate each instructions or characters into a list of tokens, we can split them by space and new-line.

After finished separating characters, scan through the list and check the type. The output is a list of tuple, each tuple has ‘type’ and ‘value’.

In the working code, scanner is in scanner.py file.

2. A parser

First, construct first and follow set, then create parsing table.

Grammar :

pgm := line pgm | EOF
line := line_num stmt
stmt := asgmnt | if | print | goto | stop
asgmnt := id = exp
exp := term exp'   ** Split exp into exp and exp’ to make it LL(1) parsable **
exp' := + term | - term | empty
** Added empty to exp’ since the example on the website use it **
term := id | const
if := IF cond line_num
cond := term cond'   ** Split cond into cond and cond’ to make it LL(1) parsable **
cond' := < term | = term
print := PRINT id
goto := GOTO line_num
stop := STOP

First Set / Follow Set :

first(pgm) = { line_num | EOF }	follow(pgm) = { EOF }
first(line) = { line_num }	follow(line) = { line_num | EOF }
first(stmt) = { id | IF | GOTO | STOP }	follow(stmt) = { line_num | EOF }
first(asgmnt) = { id }	follow(asgmnt) = { line_num | EOF }
first(exp) = { id | const }	follow(exp) = { line_num | EOF }
first(exp’) = { + | - | empty }	follow(exp’) = { line_num | EOF }
first(term) = { id | const }	follow(term) = { line_num | EOF }
first(if) = { IF }	follow(if) = { line_num | EOF }
first(cond) = { id | const }	follow(cond) = { line_num }
first(cond’) = { < | = }	follow(cond’) = { line_num }
first(print) = { PRINT }	follow(print) = { line_num | EOF }
first(goto) = { GOTO }	follow(goto) = { line_num | EOF }
first(stop) = { STOP }	follow(stop) = { line_num | EOF }

Parsing Table :

1 pgm := line pgm
2 pgm := EOF
3 line := line_num stmt
4 stmt := asgmnt
5 stmt := if
6 stmt := print
7 stmt := goto
8 stmt := stop
9 asgmnt := id = exp
10 exp := term exp'
11 exp' := + term
12 exp' := - term
13 exp' := empty
14 term := id
15 term := const
16 if := IF cond line_num
17 cond := term cond'
18 cond' := < term
19 cond' := = term
20 print := PRINT id
21 goto := GOTO line_num
22 stop := STOP

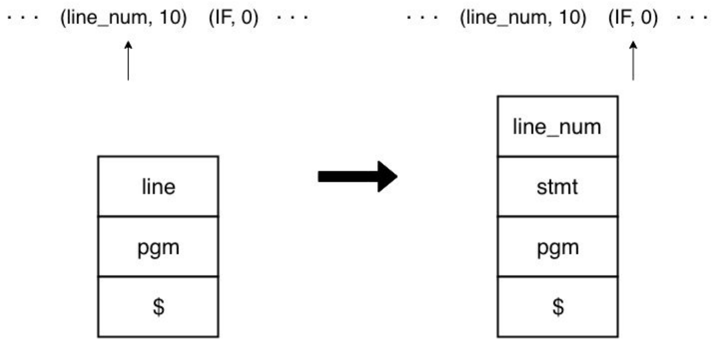
	line_num	id	const	IF	GOTO	PRINT	STOP	+	-	<	=	EOF
pgm	1											2
line	3											
stmt		4		5	7	6	8					
asgmnt		9										
exp		10	10									
exp’	13							11	12			13
term		14	15									
if				16								
cond		17	17									
cond’										18	19	
print						20						
goto					21							
stop							22					

id is {A..Z}
const is {0..100}
** Changed from {1..100} to {0..100} since the example on the website use 0 **
line_num is {1..1000}

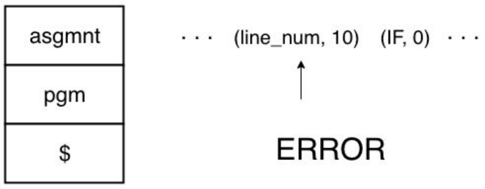
Grammar Checking:

When parsing, the parser will check if the current pointing token are in the top of stack's parsing table. If not, then it is grammatically wrong.

For example, if the top of stack is 'line' and pointing token is type 'line_num', which is in 'stmt' column in the parsing table. 'stmt' will be popped from stack and push production number 7 into the stack, which is 'line_num' and 'stmt', then move pointer forward.


. . . (line_num, 10) (IF, 0) . . .
line
pgm
$
. . . (line_num, 10) (IF, 0) . . .
line_num
stmt
pgm
$

But if the top of stack is 'asgmnt' and pointing token is type 'line_num', then it is incorrect.



3. **Pseudocode**

Scanner :

```
file = input file
tokens = []  // scanned output will be store here
for each line in file {
        temp = split each line with space
        index = 0
        while (index < length of temp) {  // iterate through split words
                if (temp[index] is number) {
                        if (0 <= temp[index] <= 100)
                                tokens.append( (line_num|const, int(temp[index])) )
                        else if (temp[index] <= 1000)
                                tokens.append( (line_num, int(temp[index])) )
                        else INVALID INPUT
                }
                else if (temp[index] is A-Z) {
                        tokens.append( (id, ascii code of temp[index] - 64) )
                }
                else if (temp[index] == IF) tokens.append((IF, 0))
                else if (temp[index] == GOTO) tokens.append((GOTO, 0))
                else if (temp[index] == PRINT) tokens.append((PRINT, 0))
                else if (temp[index] == STOP) tokens.append((STOP, 0))
                else if (temp[index] == +) tokens.append((+, 1))
                else if (temp[index] == -) tokens.append((-, 2))
                else if (temp[index] == <) tokens.append((<, 3))
                else if (temp[index] == =) tokens.append((=, 4))
                else INVALID INPUT
                index += 1  // move index forward
        }
}

return tokens
```

Parser :

```
tokens = list of tokens from source file scanning

Stack s  // stack for parsing
s.push('pgm')  // push starting symbol

pointer = 0  // pointer for tokens

output = []  // b-code output will be stored here
error = false  // error flag

while (s is not empty and pointer is not the end of tokens) {
    top = s.pop()  // pop top of stack
    if (top is terminal symbol and tokens[pointer] match top) {
        b_code = translated b-code from pointing token
        output += b_code
        pointer += 1  // move pointer forward
    }
    else {
        if (tokens[pointer] is in top's row of parsing table) {
            ps = production from parsing table
            for p each ps {
                s.push(p)  // push to top of stack
            }
        }
        else {
            error = true
            break
        }
    }
}

if (not error) {
    write output into output file
}
else {
    write error into output file
}
```

In working code, functions like b-code translating, parsing table checking will be in Grammar class in grammar.py

4. Working Code

The program is written in Python 3.7 (also runnable in Python 2.7).

There will be 3 Python files and some text files (2 of them are input and output file).

Download link

Google Drive (QR Above) : https://goo.gl/ZVq69n
OR
github.com/maxnatchanon/Retro-Basic-Compiler

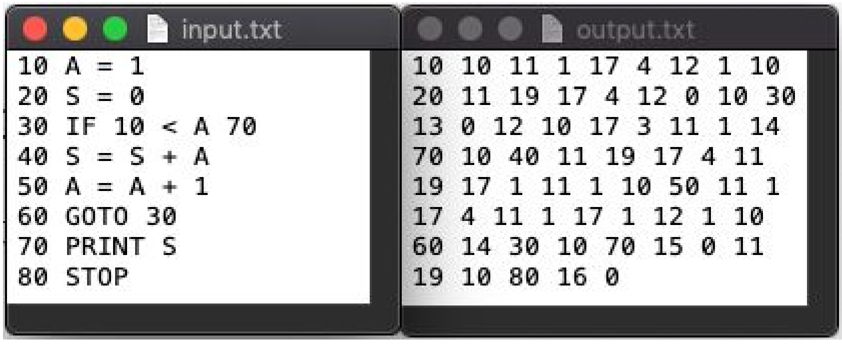
How to run :

Put your input in *input.txt*

Run *main_compiler.py*

The output will be in *output.txt*

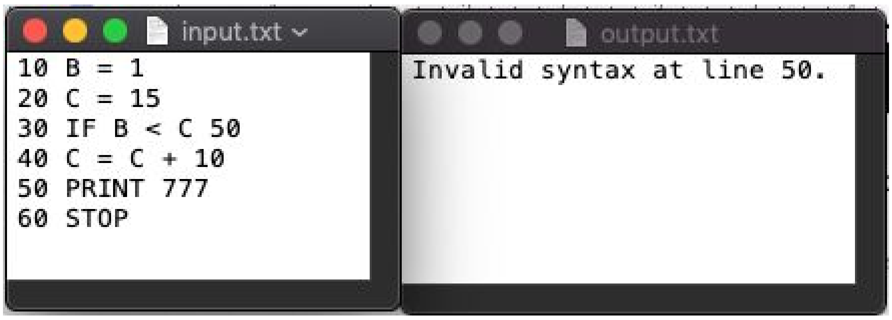
Output example :



```
10 A = 1
20 S = 0
30 IF 10 < A 70
40 S = S + A
50 A = A + 1
60 GOTO 30
70 PRINT S
80 STOP
```

```
10 10 11 1 17 4 12 1 10
20 11 19 17 4 12 0 10 30
13 0 12 10 17 3 11 1 14
70 10 40 11 19 17 4 11
19 17 1 11 1 10 50 11 1
17 4 11 1 17 1 12 1 10
60 14 30 10 70 15 0 11
19 10 80 16 0
```

Parsing successful



```
10 B = 1
20 C = 15
30 IF B < C 50
40 C = C + 10
50 PRINT 777
60 STOP
```

```
Invalid syntax at line 50.
```

Parsing error

(Actual working code available on the last page)