BRIEF EXPLANATION ABOUT HOW THE WHOLE PROGRAM WORKS

If the inputFile is valid, it Lexical Analyze the code. If Lexical Analyzer does not encounter LA mistakes, it creates a la_out array. Parse takes it and writes it to a output array, if parser too does not encounter any Parsing errors output will be written into console.

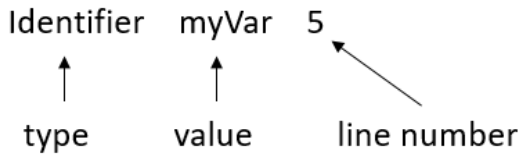
```
LexicalAnalyze(inputFile);
Parse(la_out);

int i;
for (i = 0; i < output_index + 1; i++)
{
    if (i != output_index)
        printf("%s\n", output[i]);
    else
        printf("%s", output[i]);
}

return 0;
```

Variables and its' value's exist inside two seperate arrays.

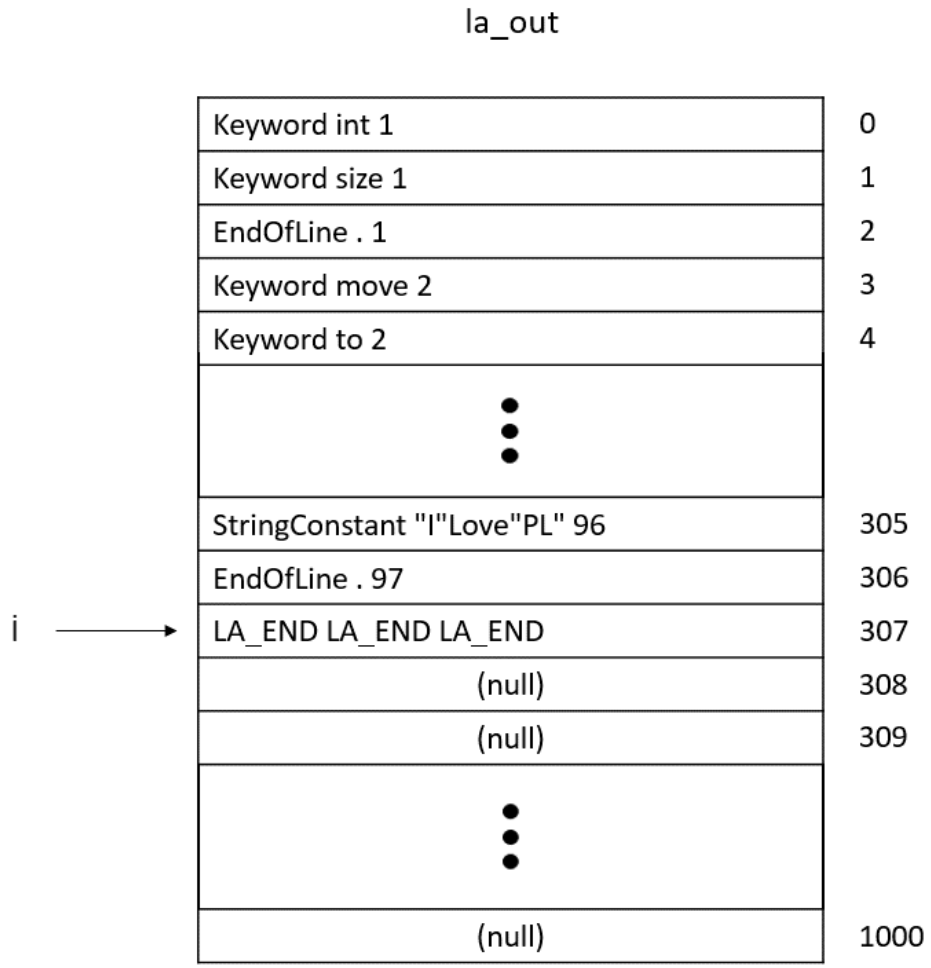
For loop operations, loop's start and finish indexes exist. And they work on la_out array. If code is loop -3 times alike, loop works for 1 time but do not write any data to console, just checking if there is any mistake.

CHANGES HAPPENED TO LEXICAL ANALYZER

Lexical analyzer's information format has changed. Line number added to it's previous format for precise errors.



Lexical analyzer still gets input from a file, but it's output is no longer a file. String array



NOTE: In the la_out's visual at line 305, StringConstant "I love PL" held as "I"love"PL". The reason behind it is that, when parser reads la_out's lines, it uses sscanf in the format shown below.

```
sscanf(*input[i++],"%s %s %s", &type, &value, &line_num_s);
```

If la_out helds it as StringConstant "I love PL" 305,

type = StringConstant
value = "I
line_num_s = love PL" 305 will happen.

In order to avoid that spaces becomes quotation marks. And it will look like this.

type = StringConstant
value = "I"love"PL"
line_num_s = 305.

When parser needs to use this quotation mark version, values' first and last index will be removed and value will become I"love"PL. After that, with a for loop quotation marks will be changed with space again. And value will be printed like I love PL.

MODIFICATION 1 (CASE-INSENSITIVITY)

In order to make keywords and identifiers case-insensitive, there is a new function called "isIdentifier" in Lexical Analyzer.

```
case 1: // reading identifier or keyword
    ch = fgetc(inputFile);

    if (!isalnum(ch) && ch != '_')
    {
        word[i] = '\0';

        if (isIdentifier(word))
            strcat(caseout, "Identifier ");
        else if (isKeyword(word))
            strcat(caseout, "Keyword ");
```

isIdentifier, checks the word if it is identifier or not but does not not change the original word unlike isKeyword.

for example:

mOvE -----> isIdentifier -----> mOvE -----> isKeyword -----> **move**
myVariabLe -----> isIdentifier -----> **myVariabLe**

The reason isIdentifier does not change the original word is in the test statement, console output must be the exact in written.

```
Test myVariable.

{ should display the line

  myVariable:25

and a newline on the screen}
```

So far, keywords are case-insensitive. For identifiers, in the Parse function, variables are stored with all lower case characters. So when Parser needs to check if variable exists, it converts current identifiers' name to all lower case characters, then compares.

```c
int isVarDeclared(char s[])
{
    char c[20];
    strcpy(c, s);

    int j;
    for (j = 0; c[j]; j++)
        c[j] = tolower(c[j]);

    int i;
    for (i = 0; i < 100; i++)
        if(strcmp(c, var_list[i]) == 0)
            return 1;

    return 0;
}
```

MODIFICIATION 2 (C-STYLE COMMENTS)

Slash (/) and asteriks (*) are now valid characters for the interpreter. When L.A. sees the slash at state 0 (state determiner), it checks the next character. If the next character is also a slash that means it is a multi line comment, else if the next character is an asteriks that means it is a single line comment. Then, the new state is 4 (comment state).
In the comment state, if it is //, when a newline occurs state will be 0. If it is /*, it will check for *, when it's found it will check the next character, if it is / state will be 0, else will continue to seek * which has a next char as /.

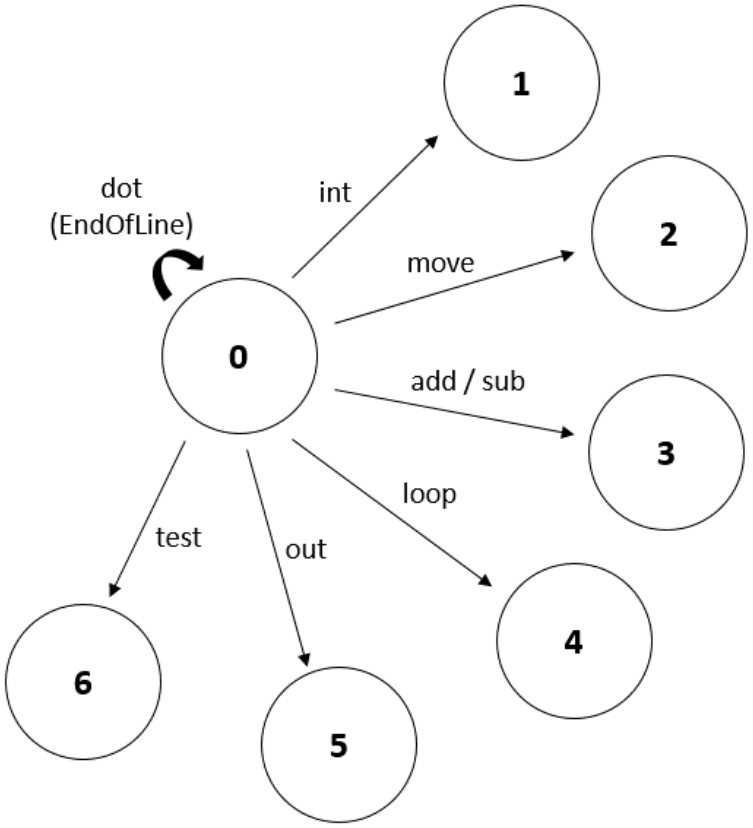
In state 0:

```
if (ch == '/')
{
    ch = fgetc(inputFile);
    if (ch == '*' || ch == '/')
    {
        curlyComment = 0;
        commentIsClosed = 0;
        state = 4;
        if (ch == '*')
            multiLineComment = 1;
        else
            multiLineComment = 0;

        break;
    }
```

In state 4:

```
// ignoring everything unless ch is }
if (ch == '}' && curlyComment)
{
    commentIsClosed = 1;
    state = 0;
    break;
}

// ignoring everything unless sees newline
if (ch == '\n' && !curlyComment && !multiLineComment)
{
    commentIsClosed = 1;
    multiLineComment = 1;
    state = 0;
    break;
}

// ignoring everything unless sees */
if (ch == '*' && !curlyComment && multiLineComment)
{
    ch = fgetc(inputFile);
    if (ch == '/')
    {
        commentIsClosed = 1;
        state = 0;
        break;
    }
```

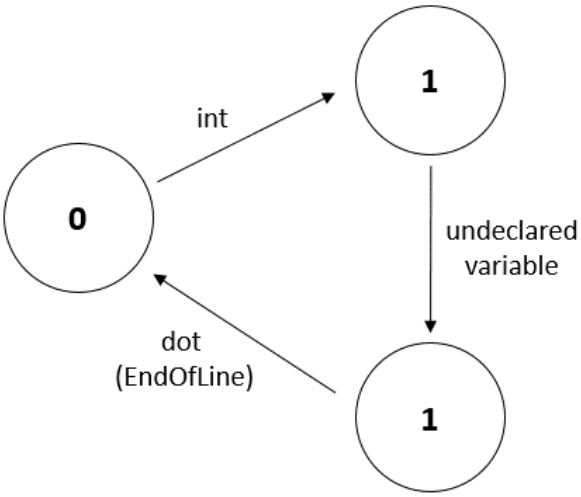
STATE 0: DETERMINATION OF THE STATES



Apart from OpenBlocks and CloseBlocks, state 0 roughly works like in the picture. When it reaches "LA_END" it returns the void function Parse.

In state 0, if CloseBlock number is more than OpenBlock number it gives instant error. And in the end of the code if OpenBlock and CloseBlock counts do not match it gives error too.

STATE 1: VARIABLES

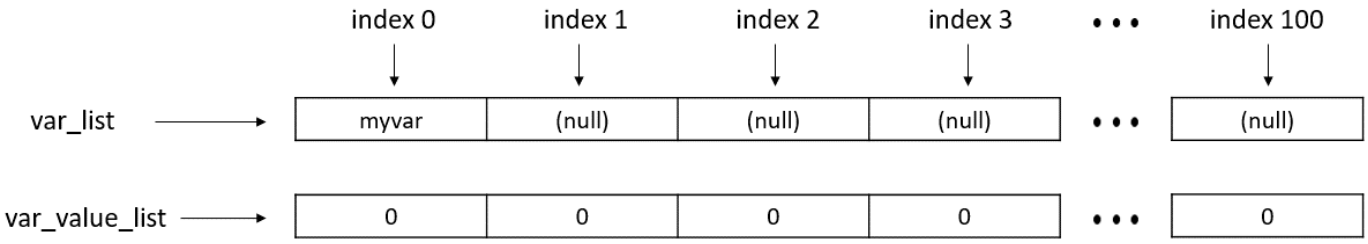


int <variable>.

When a variable gets declared, it will be added to an 1D array called var_list. And its' value will be 0 automatically and will be also kept in a 1D array called var_value_list.

Undeclared variable is a case-insensitive identifier which has a maximum length of 20 characters.

example: int myVar.



myVar will be stored at var_list as myvar because of case-insensitivity. For example, if code includes MyVaR, it will get lowercased and myvar will be checked if it exists in var_list.

FUNCTIONS FOR VARIABLE HANDLING

The parser has 3 functions called "isVarDeclared", "getVarValue", "setVarValue" for var_list and var_value_list.

1) isVarDeclared

isVarDeclared will take a string, copies it to an another string with all lowercase characters and then checks if it exists in var_list.

```
int isVarDeclared(char s[])
{
    char c[20];
    strcpy(c, s);

    int j;
    for (j = 0; c[j]; j++)
        c[j] = tolower(c[j]);

    int i;
    for (i = 0; i < 100; i++)
        if(strcmp(c, var_list[i]) == 0)
            return 1;

    return 0;
}
```

2) getVarValue

getVarValue will check if the variable name exists in var_list just like isVarDeclared. If it exists it will get the value from var_value_list at the same index it founds var exists.

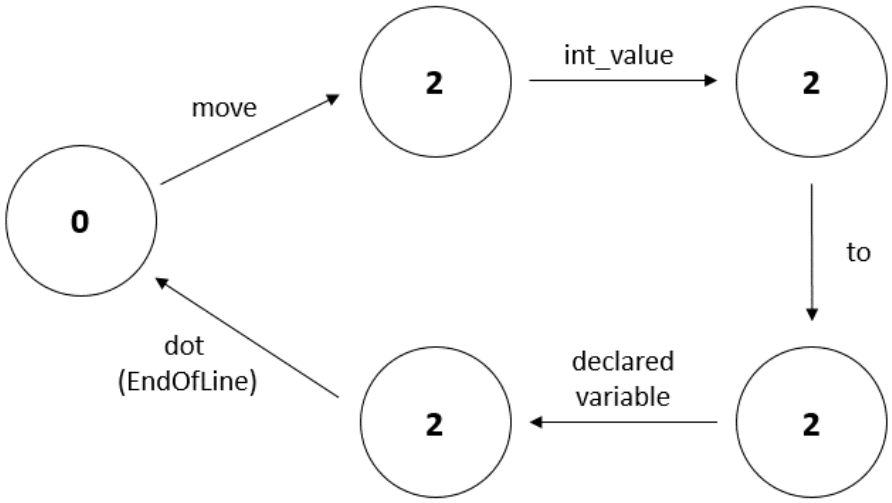
```
int getVarValue(char s[])
{
    char c[20];
    strcpy(c, s);

    int j;
    for (j = 0; c[j]; j++)
        c[j] = tolower(c[j]);

    int i;
    for (i = 0; i < 100; i++)
        if (strcmp(c, var_list[i]) == 0)
            break;

    return var_value_list[i];
}
```

3) setVarValue

setVarValue takes int value other than two functions. It checks if variable name exists in var_list like the other two functions. If it finds, it will change the current value from var_value_list at the same index it founds var exists to the new value.

```
void setVarValue(char s[], int value)
{
    char c[20];
    strcpy(c, s);

    int j;
    for (j = 0; c[j]; j++)
        c[j] = tolower(c[j]);

    int i;
    for (i = 0; i < 100; i++)
        if (strcmp(c, var_list[i]) == 0)
            break;

    var_value_list[i] = value;
}
```

STATE 2: ASSIGMENT

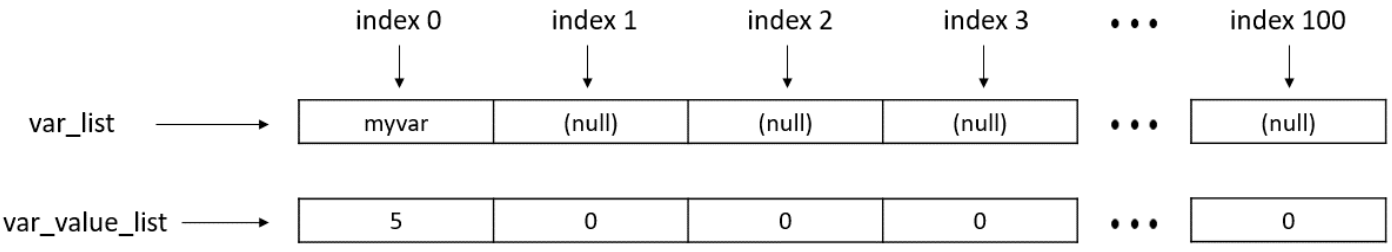


move <int value> to <variable>.

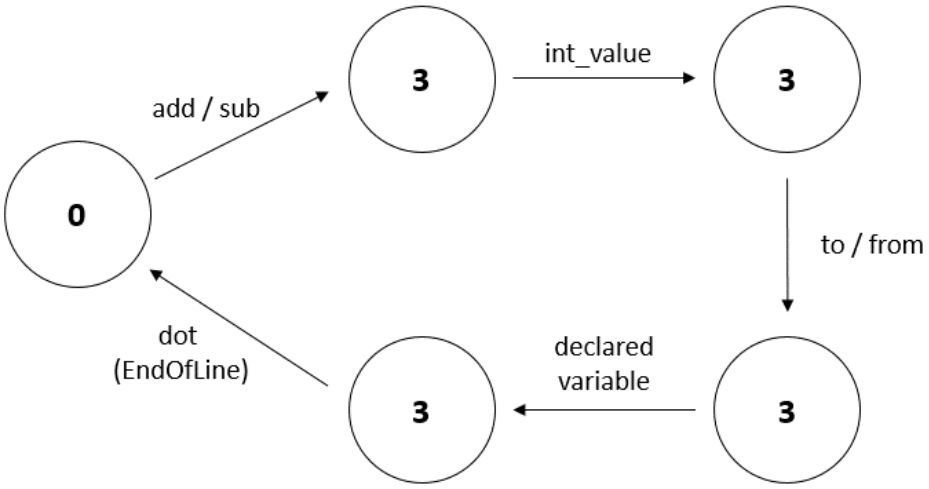
If int_value is IntConstant, setVarValue function will change the current value from var_value_list to new value.

Else if int value is Identifier, isVarDeclared will check that it exists in var_list, getVarValue will get the value from var_value_list at the same index it founds variable exists in var_list, setVarValue will change the current value to the new value.

example: mOvE 5 to MyVAR.



STATE 3: ADDITION / SUBTRACTION

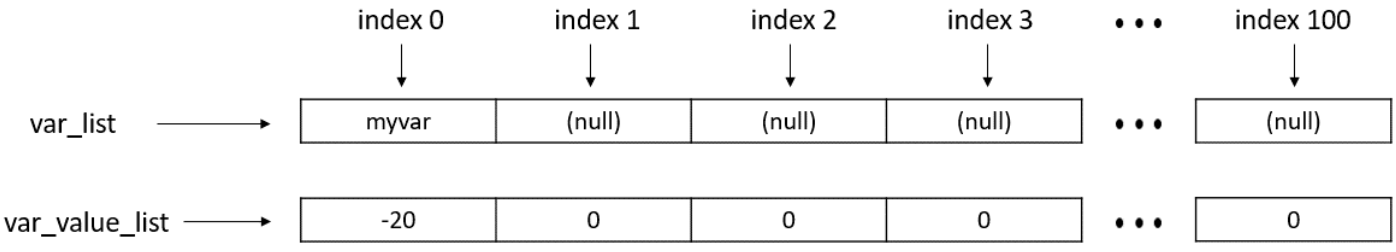


addition: add <int_value> to <variable>.

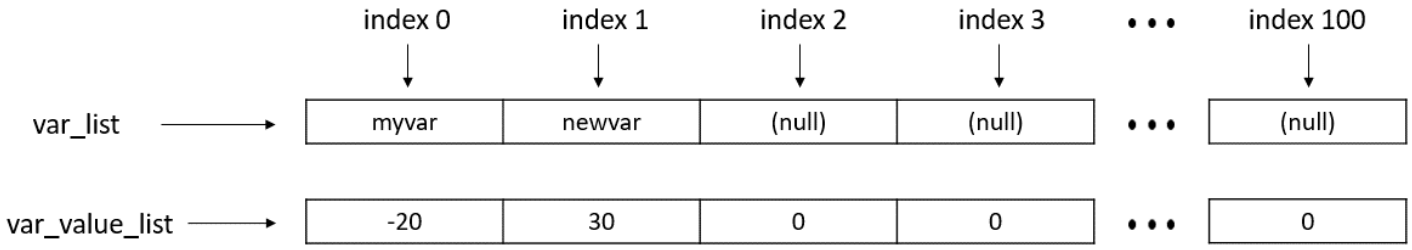
subtraction: sub <int_value> from <variable>.

If int_value is valid and variable at the righmost is in the var_list, var_value_list will change due to int_value.

example: aDD -25 to MYvar.



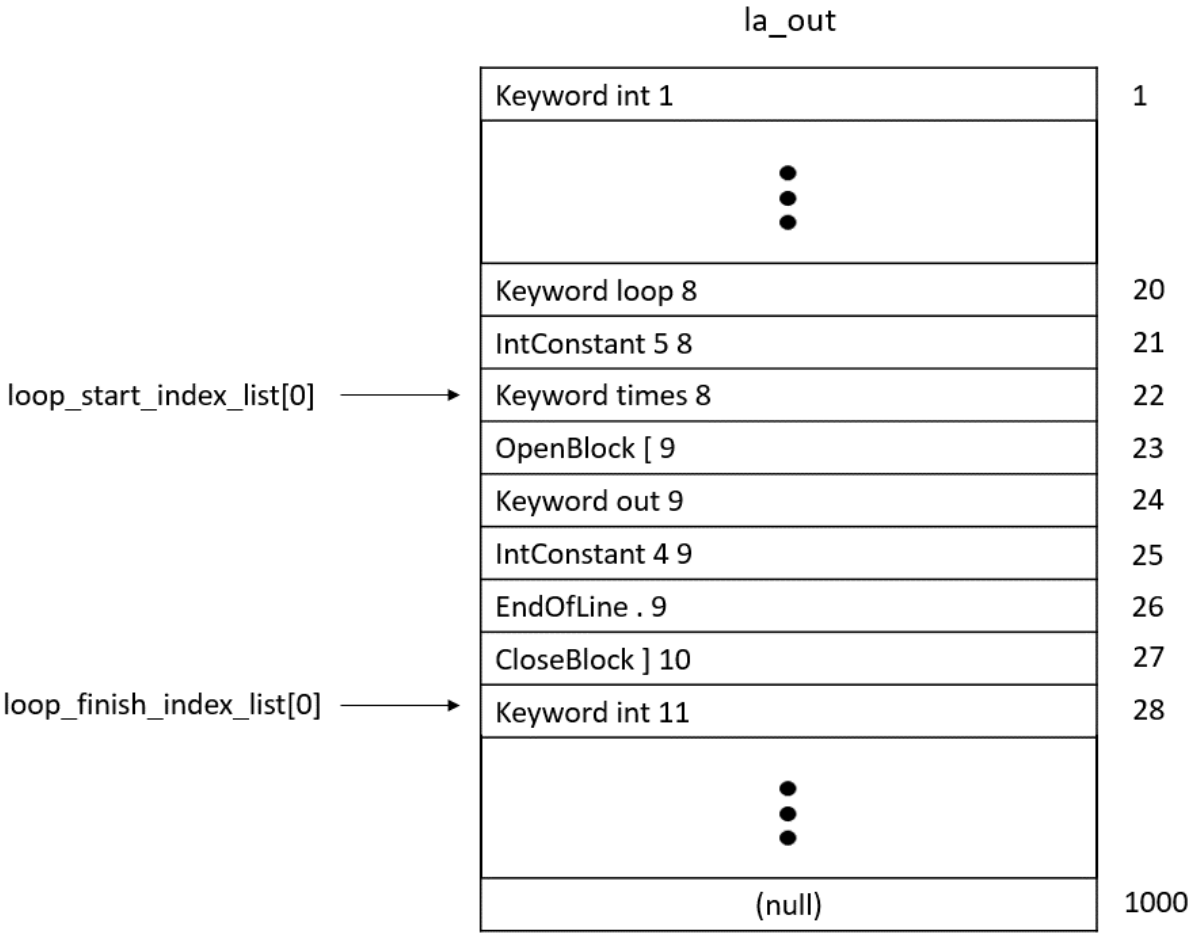
example: int newvar. SUB -30 from NewVar.



STATE 4: LOOP

If a loop starts with OpenBlock, OpenBlock's index + 1 on the la_out is where the loop starts. Else if the loop starts with other than OpenBlock, keyword time's index + 1 on the la_out is where the loop starts. There is a 1D array in the code called loop_start_index_list to keep these indexes at memory. Reason, because it is not a integer but array is for nested loop handling.

Like loop_start_index_list, there is a loop_finish_index_list too. For OpenBlock type of loop, finish index is CloseBlock's index + 1. For the other type, finish index is EndOfLine's index + 1. The reason because plus one is that parser will start the read from index + 1.



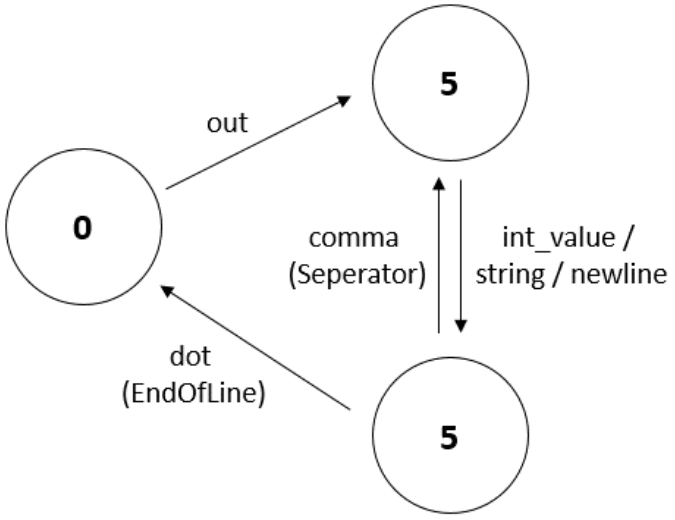
In a for loop recursive Parse function will take la_out as an input and with the indexes from loop index lists, the program will do its job. If int_value of the for loop is a variable, variable's value will be decremented by 1.

```
int la;
for (la = 0; la < loop_index; la++)
{
    if (var_loop)
        setVarValue(index_name, loop_index - la);

    Parse(la_out);
}

if (var_loop)
    setVarValue(index_name, 0);
```

If loop index is less than 1, it gets 1 value and join the for loop but do not process the data.

There is one more list called loop_complete_list, it uses the same indexes with loop_start_index_list and loop_finish_index_list. It's all values are 0 at the beginning. If loop complated it takes 1, for nested loops it is needeed. When loop 4 times loop 3 times loop 2 times out 3. for example, first 2 loops will be done. Then it will check if the 3 loop happened by loop_complete_list, if not will done the 3 loop then 4 loop then it is okay.

STATE 5: OUT



<out_list> → <out_list>,<list_element>|<list_element>

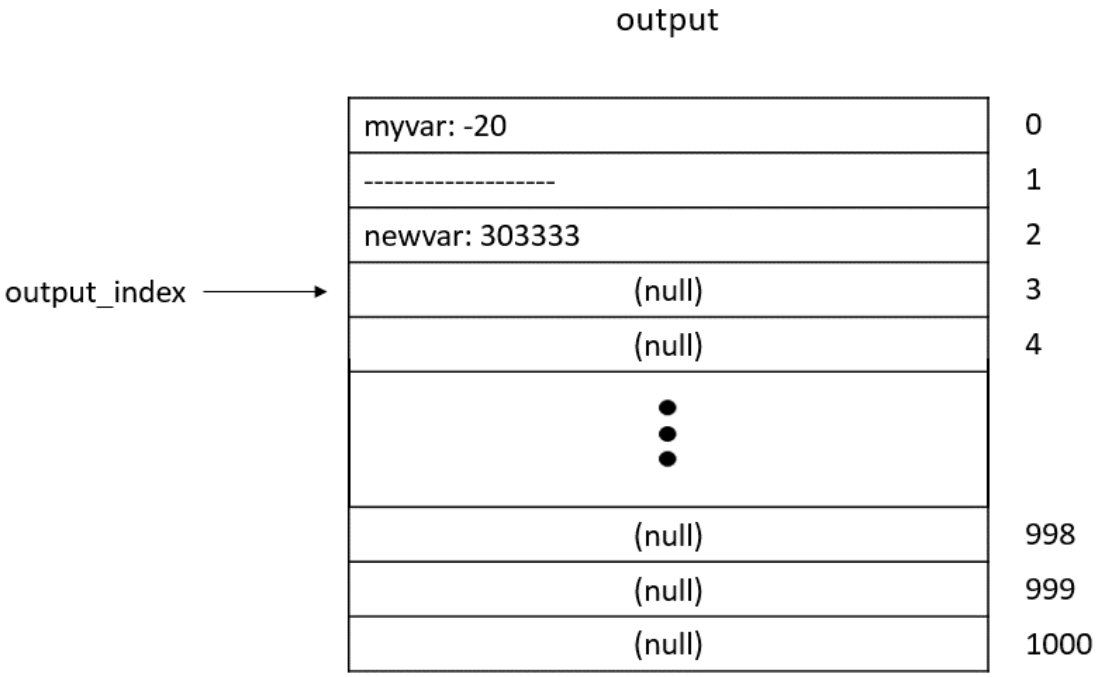
<list_element> → <int_value>|<string>| newline

If int_value or string comes after the out keyword, it will be written to a string called output_line. This output_line will be written into a string array called output. When newline appears, output_line will be kept as final at where output_index shows in output array. After that, output_index will be incremented by 1 and output_line will be empty. This will go on like that.

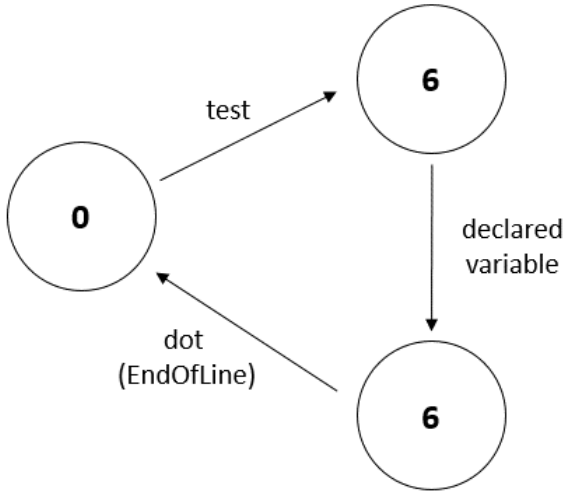
When parser finishes it's job, output array will be print out line by line until it reaches output_index's last value.

example:

out "myvar: ", myvar, newline.
Out "----------", newline.
OUT "newvar: ", newvar.
Out 3333, newline.



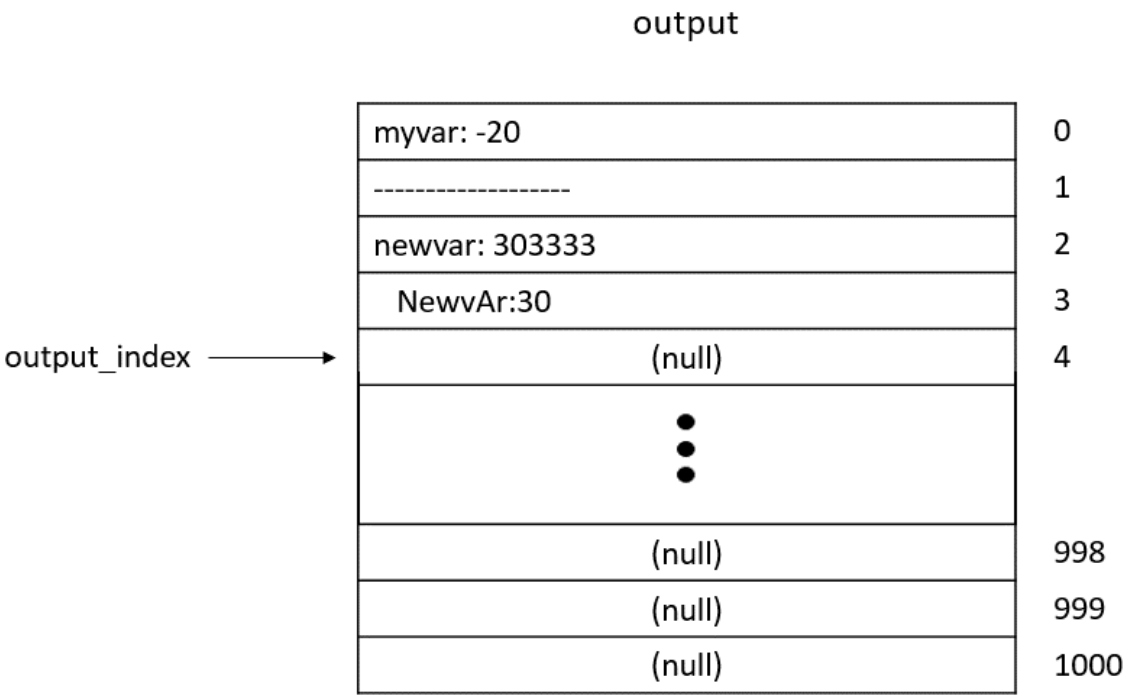
STATE 6: TEST / MODIFICATION 3



test <variable>.

Like the out statement, variable's value is written to output. But in a formatted way with help of strcat and strcpy functions of C.

example: TesT NewvAr.



RUN EXAMPLE 1 (FOR LOOP, CASE-INSENSITIVITY, C-STYLE COMMENTS)

INPUT:

```
INt SiZe.
int SuM.
// this is a comment
MOVe 5 tO SIZE. /* this is also a comment
*/
lOOp size TIMES
[ OUT sIze, Newline.
ADd Size to sum.
]
oUT neWLiNe, "Sum:", sUm.
```

OUTPUT:

```
5
4
3
2
1

Sum:15
```

RUN EXAMPLE 2 (TEST STATEMENT)

INPUT:

```
int SIZE.
int sum.
move 8 to SIzE.
loop size times
[ out sIzE.
add sizE to sum.
TEST SUM.
]
out newline, "Sum:", sum.
out newline.
test size.
```

OUTPUT:

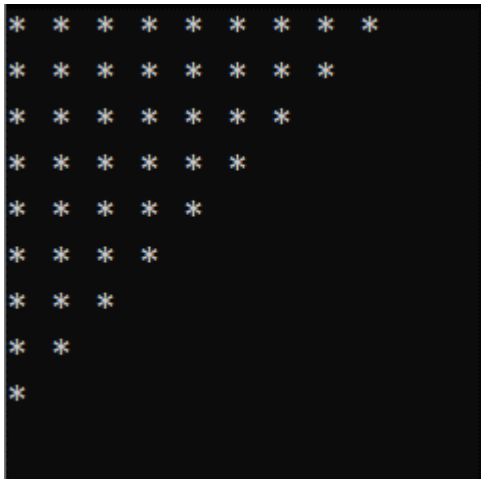
```
8
  SUM:8
7
  SUM:15
6
  SUM:21
5
  SUM:26
4
  SUM:30
3
  SUM:33
2
  SUM:35
1
  SUM:36

Sum:36

  size:0
```

RUN EXAMPLE 3 (NESTED LOOPS)

INPUT:

```
INT I.
INT J.
move 9 to i.
Loop i times
[
   Move i to j.
   Loop j times [
     OUT "* ".
   ]
   OUT neWline.
]
```

OUTPUT:



RUN EXAMPLE 4 (LOOP HEAVEN)

INPUT:

```
out newline. /* just passing through */ { cuz im free }
int size.
move 5 to size.
loop size times
[
loop 2 times loop 4 times loop 3 times out 6.
out " ----- ".
loop 3 times [ out 8. ]
out " ----- ".
loop 4 times out 7.
out " ----- ".
loop -8 times loop 2 times out 0. // it wont write
test size.
]
out newline.
test size.
loop 9 times loop 2 times out 1.
loop -1 times out 3.
out 78.
```

OUTPUT:

```
555
666666666666666666666666 ----- 888 ----- 7777 -----
  size:5
666666666666666666666666 ----- 888 ----- 7777 -----
  size:4
666666666666666666666666 ----- 888 ----- 7777 -----
  size:3
666666666666666666666666 ----- 888 ----- 7777 -----
  size:2
666666666666666666666666 ----- 888 ----- 7777 -----
  size:1


  size:0
11111111111111111178
```

RUN EXAMPLE 5 (SPELLING MISTAKE FOR ADD STATEMENT)

INPUT:

```
int a.
int b.
add a from b.
out a, newline.
out b.
```

OUTPUT:

```
[Error] 'to' expected after 'add' at line 3
```

RUN EXAMPLE 6 (VARIABLE ALREADY DECLARED)

INPUT:

```
int a.
int a.
int b.
move 4 to a.
add a to b.
out a.
```

OUTPUT:

```
[Error] 'a' at line 2 is already declared
```

RUN EXAMPLE 7 (COMMENT LEFT OPEN)

INPUT:

```
/*
int int int int
here comes the rain again
aaaahhhhhh
```

OUTPUT:

```
[Error] CommentLeftOpen at line 4
```

RUN EXAMPLE 8 (TEST INTEGER)

INPUT:

```
int a.
int b.
move  1 to a.
test 3.
test a.
test b.
```

OUTPUT:

```
[Error] variable expected after 'test' at line 4
```