SER502-SPRING2018-TEAM<24>

Hongfei Ju
Zachary Wang
Ruihao Zhou

outline

- Language name
- Grammar rule
- Compiler
- Lexcial analyzer
- Parser
- Intermediate code generator
- Runtime

Language description

Name: L0 (language zero)

Zero is the first natural number, we see L0 as a good start of our learning in compiling techniques.

Language description

Primitive Type:

boolean values and int numeric value.

Operation:

For boolean type data: “equal”, “larger than”, “no less than”, “less than”, no larger than”, “not equal”.

For int numeric type data: “plus”, “minus”, “multiply”, “divide”.

Statements:

assignment to associate a value with a variable

if-then-else statement to make decisions

while statement for iterative execution

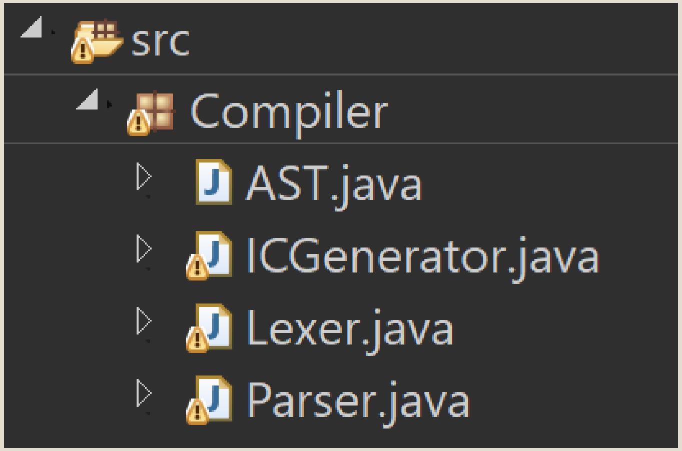
Grammar rule

```
program  :  statement_list ;

statement_list  :  statement  statement_list  |  statement  ;

statement  :  declaration  |  assignment  |  if_statement  |  while_statement  |  print  ;

declaration  :  'var'  ID  ';'  ;

assignment  :  ID  ':='  low_expression ';'  ;

if_statement  :  'if'  '(' boolean_expression  ')'  'correct'  '{'  statement_list  '}'  |  'if'  '(' boolean_expression  ')'  'correct'  '{'  statement_list  '}'  'wrong'  '{'  statement_list  '}'  ;

while_statement  :  'while'  '('  boolean_expression  ')'  '{'  statement_list  '}'  ;

print  :  'print'  low_expression  ;
```

Grammar rule

```
boolean_expression  :  low_expression  '=='  low_expression  |  low_expression  '>'
low_expression  |  low_expression  '>='   low_expression  |  low_expression '<'
low_expression  |  low_expression  '<='  low_expression  |  low_expression  '!='
low_expression  |  boolean_val  ;

boolean_val  :  'true'  |  'false'  ;

low_expression  :  high_expression  '+'  low_expression  |  high_expression  '-'
low_expression  |  high_expression  ;

high_expression  :  item  '*'  high_expression  |  item  '/'  high_expression  |  item  ;

item  :  ID | NUMBER  ;

ID  :  [a-z|A-Z]+  ;

NUMBER  :  [0-9]+  ;

WS :  [ \t\r\n]+ -> skip ;
```

Compiler


src
Compiler
AST.java
ICGenerator.java
Lexer.java
Parser.java

Source code

Lexical analyzer

Token list

Parser

Parse tree

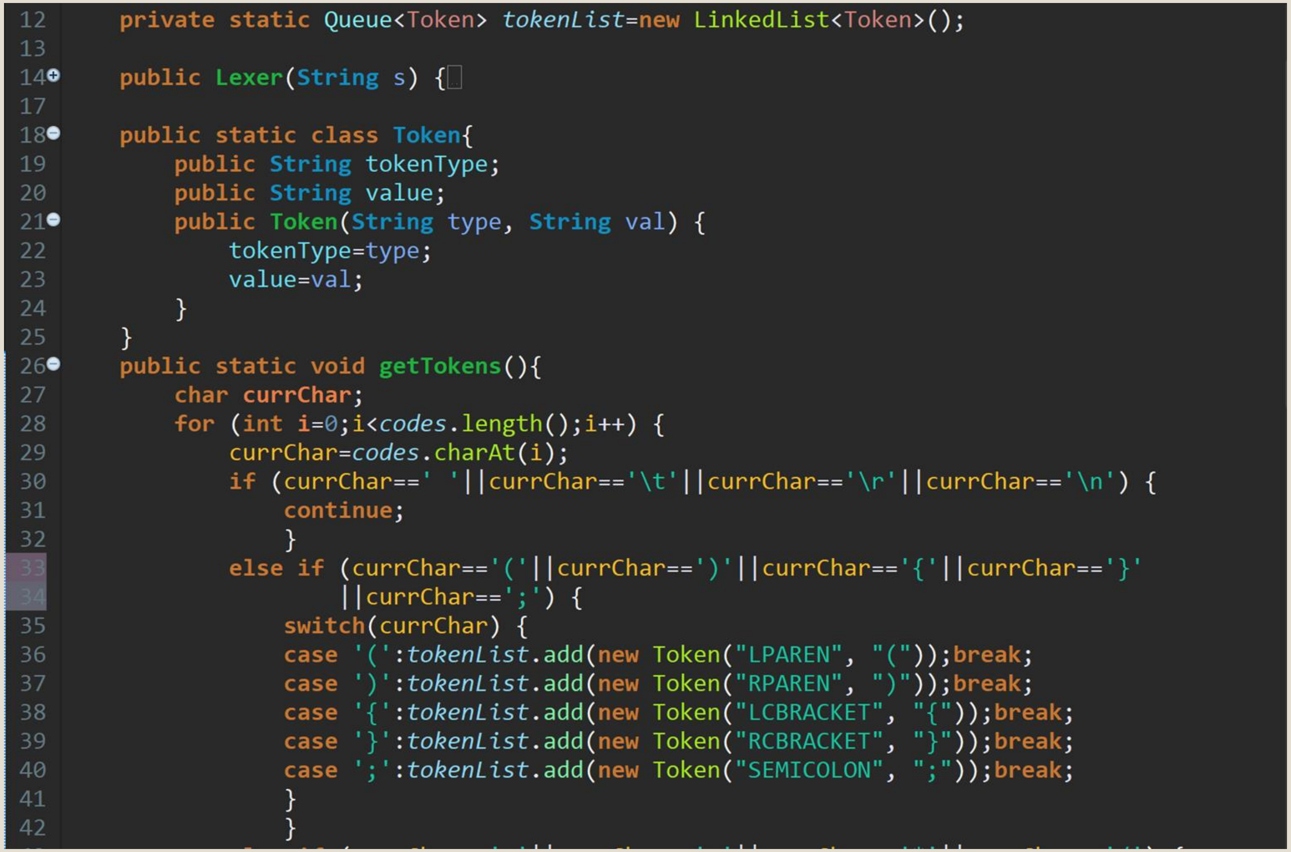
Intermediate code generator

Intermediate code

Lexical analyzer

Data structure: symbol table (token type and token value)

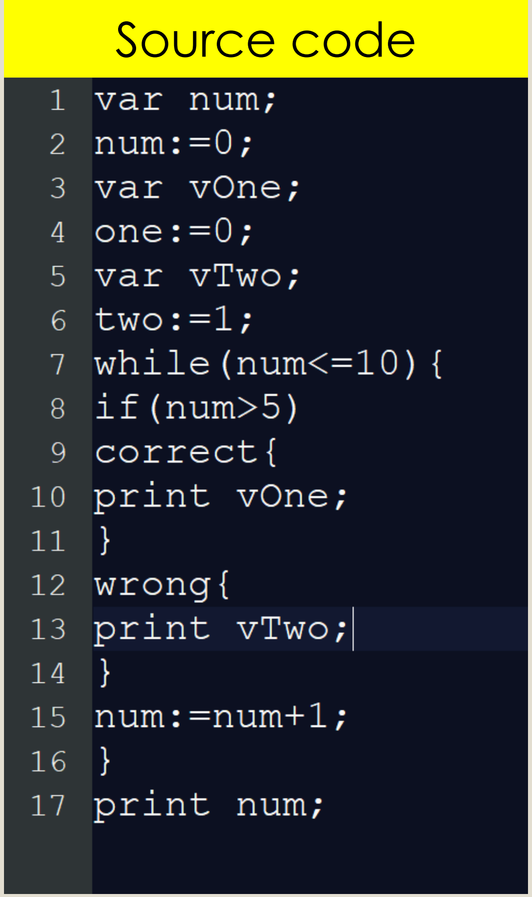
Lexical analysis： 1) read the source codes and separate the characters into tokens; 2) store the tokens and the corresponding types in the symbol table.



```java
    private static Queue<Token> tokenList=new LinkedList<Token>();

    public Lexer(String s) {

    public static class Token{
        public String tokenType;
        public String value;
        public Token(String type, String val) {
            tokenType=type;
            value=val;
        }
    }
    public static void getTokens(){
        char currChar;
        for (int i=0;i<codes.length();i++) {
            currChar=codes.charAt(i);
            if (currChar==' '||currChar=='\t'||currChar=='\r'||currChar=='\n') {
                continue;
                }
            else if (currChar=='('||currChar==')'||currChar=='{'||currChar=='}'
                    ||currChar==';') {
                switch(currChar) {
                case '(':tokenList.add(new Token("LPAREN", "("));break;
                case ')':tokenList.add(new Token("RPAREN", ")"));break;
                case '{':tokenList.add(new Token("LCBRACKET", "{"));break;
                case '}':tokenList.add(new Token("RCBRACKET", "}"));break;
                case ';':tokenList.add(new Token("SEMICOLON", ";"));break;
                }
                }
```

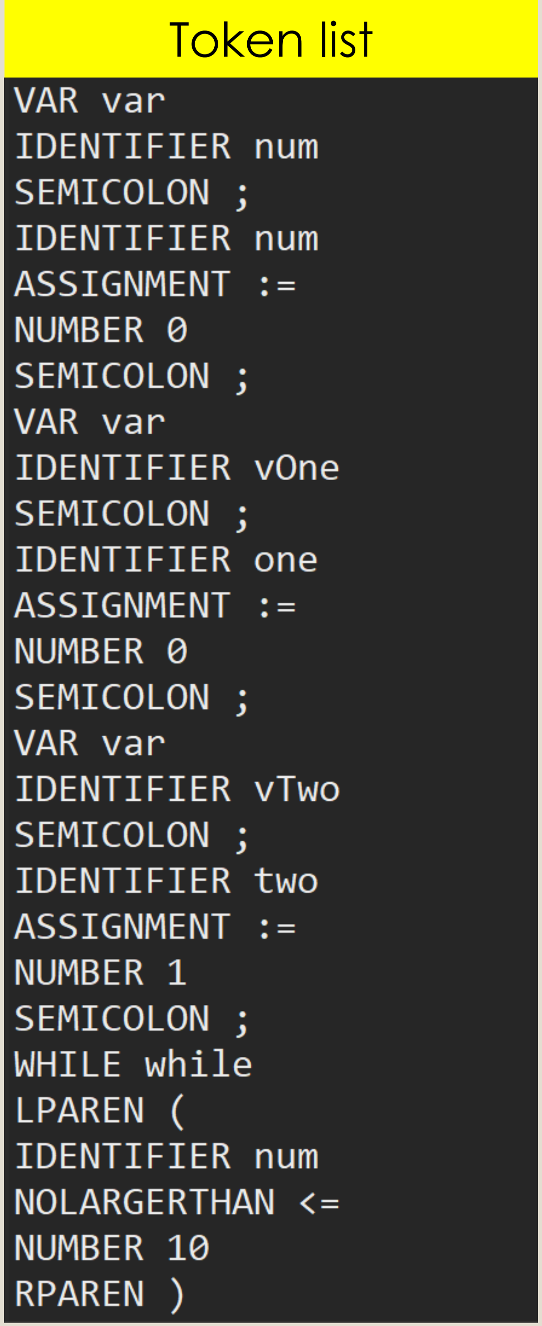
Lexical analyzer

Token list example



Source code

```
var num;
num:=0;
var vOne;
one:=0;
var vTwo;
two:=1;
while(num<=10){
if(num>5)
correct{
print vOne;
}
wrong{
print vTwo;
}
num:=num+1;
}
print num;
```



Token list

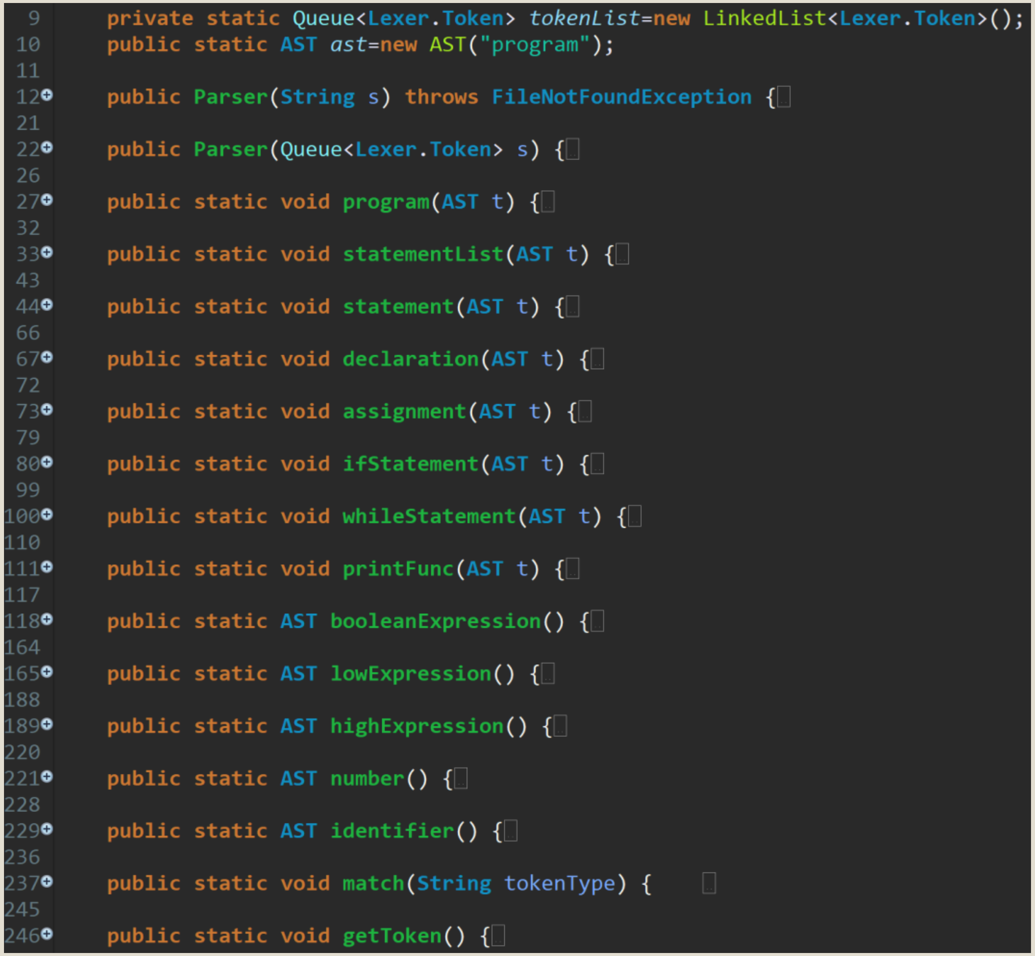
```
VAR var
IDENTIFIER num
SEMICOLON ;
IDENTIFIER num
ASSIGNMENT :=
NUMBER 0
SEMICOLON ;
VAR var
IDENTIFIER vOne
SEMICOLON ;
IDENTIFIER one
ASSIGNMENT :=
NUMBER 0
SEMICOLON ;
VAR var
IDENTIFIER vTwo
SEMICOLON ;
IDENTIFIER two
ASSIGNMENT :=
NUMBER 1
SEMICOLON ;
WHILE while
LPAREN (
IDENTIFIER num
NOLARGERTHAN <=
NUMBER 10
RPAREN )
```

Parser

Data structure: abstract syntax tree

Parsing:

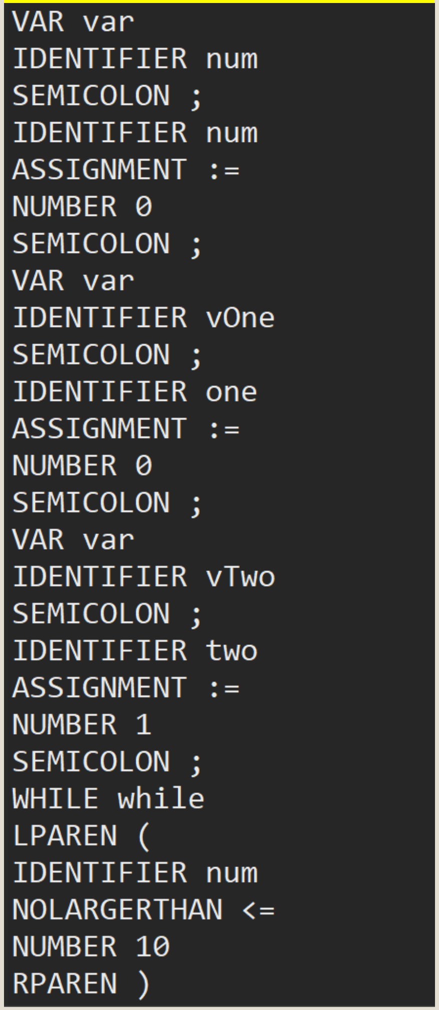
1) match the tokens with the grammar rules;
2) generate the nodes on the parse.



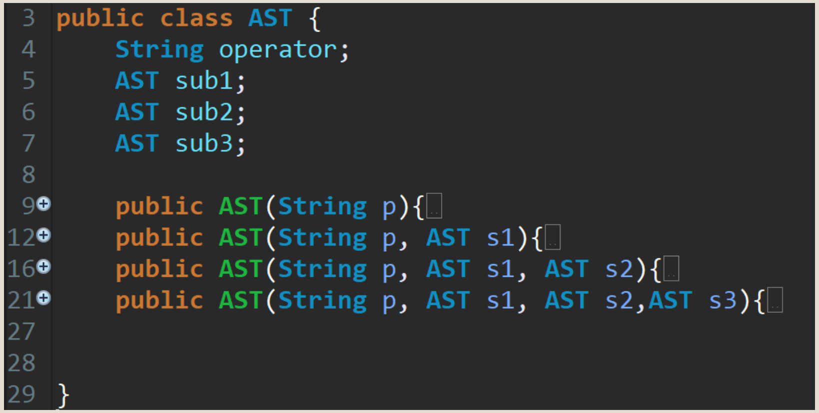
```
    private static Queue<Lexer.Token> tokenList=new LinkedList<Lexer.Token>();
    public static AST ast=new AST("program");

    public Parser(String s) throws FileNotFoundException {□

    public Parser(Queue<Lexer.Token> s) {□

    public static void program(AST t) {□

    public static void statementList(AST t) {□

    public static void statement(AST t) {□

    public static void declaration(AST t) {□

    public static void assignment(AST t) {□

    public static void ifStatement(AST t) {□

    public static void whileStatement(AST t) {□

    public static void printFunc(AST t) {□

    public static AST booleanExpression() {□

    public static AST lowExpression() {□

    public static AST highExpression() {□

    public static AST number() {□

    public static AST identifier() {□

    public static void match(String tokenType) {    □

    public static void getToken() {□
```

Parser

Token list

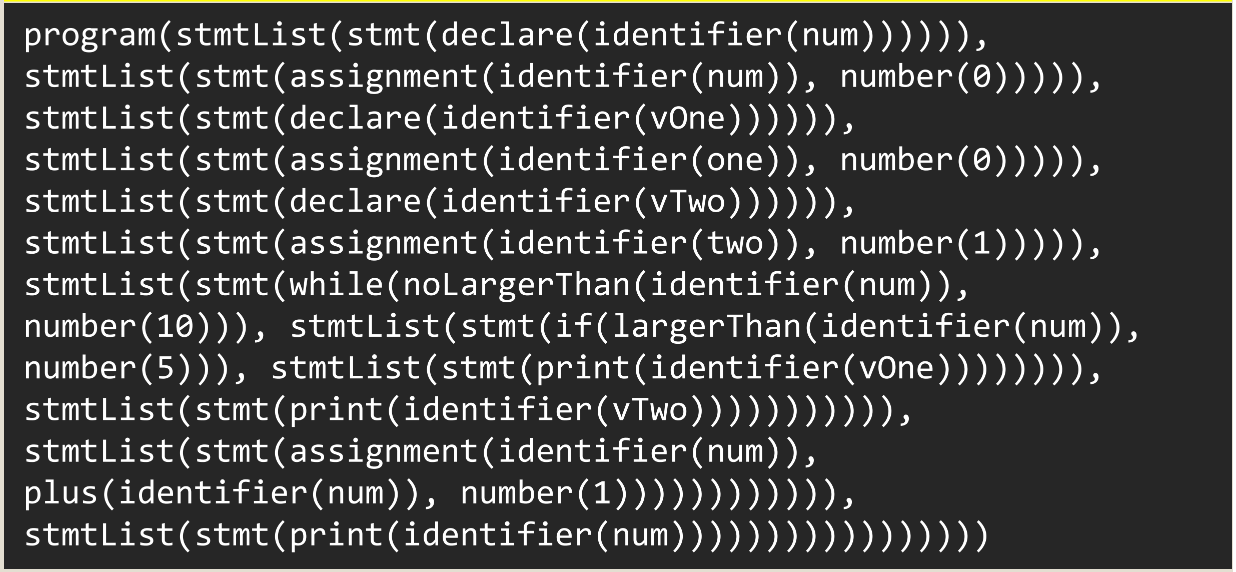


```
VAR var
IDENTIFIER num
SEMICOLON ;
IDENTIFIER num
ASSIGNMENT :=
NUMBER 0
SEMICOLON ;
VAR var
IDENTIFIER vOne
SEMICOLON ;
IDENTIFIER one
ASSIGNMENT :=
NUMBER 0
SEMICOLON ;
VAR var
IDENTIFIER vTwo
SEMICOLON ;
IDENTIFIER two
ASSIGNMENT :=
NUMBER 1
SEMICOLON ;
WHILE while
LPAREN (
IDENTIFIER num
NOLARGERTHAN <=
NUMBER 10
RPAREN )
```



```java
public class AST {
    String operator;
    AST sub1;
    AST sub2;
    AST sub3;

    public AST(String p){
    public AST(String p, AST s1){
    public AST(String p, AST s1, AST s2){
    public AST(String p, AST s1, AST s2,AST s3){


}
```

Parse tree

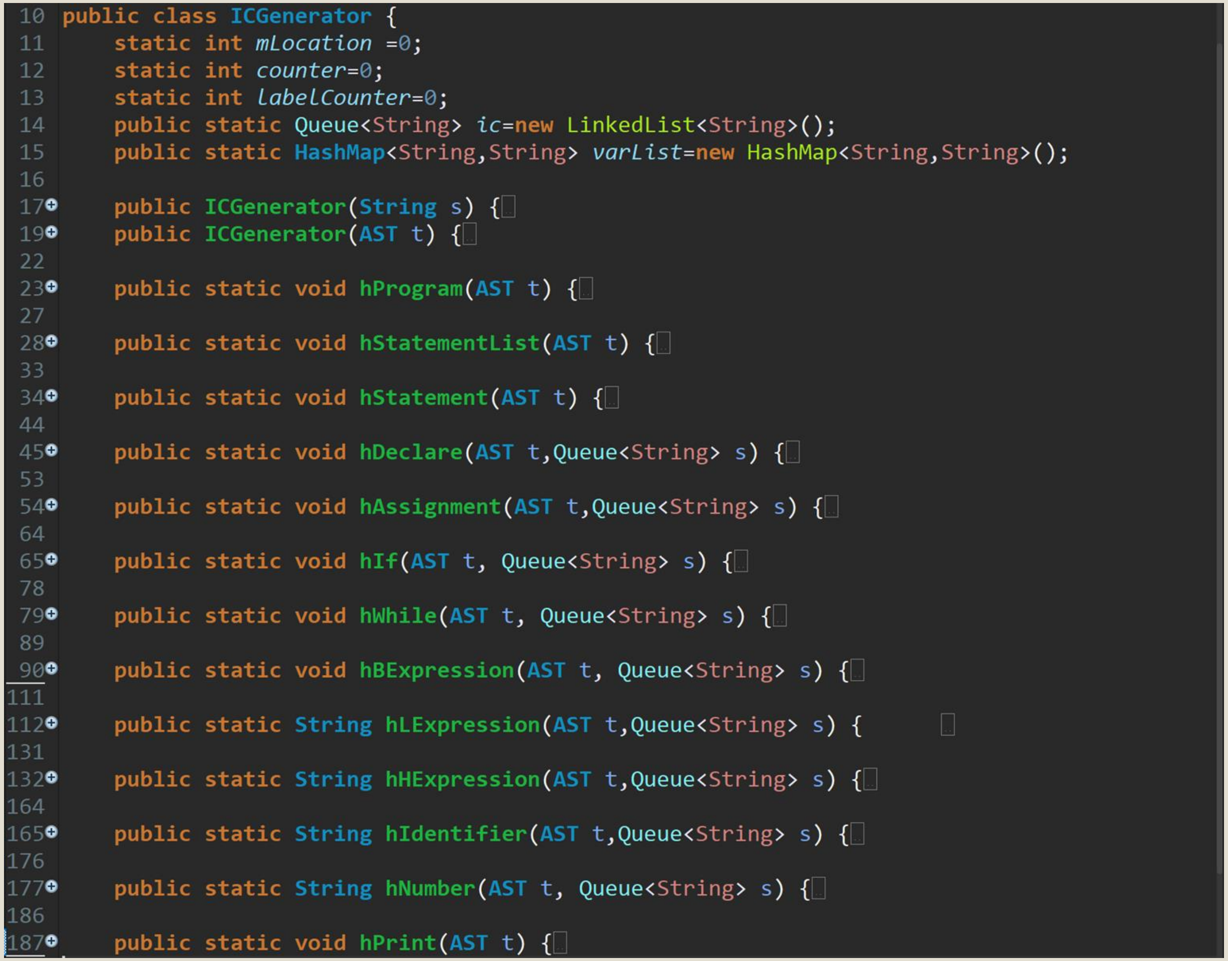


```
program(stmtList(stmt(declare(identifier(num)))))),
stmtList(stmt(assignment(identifier(num)), number(0))))),
stmtList(stmt(declare(identifier(vOne)))))),
stmtList(stmt(assignment(identifier(one)), number(0))))),
stmtList(stmt(declare(identifier(vTwo)))))),
stmtList(stmt(assignment(identifier(two)), number(1))))),
stmtList(stmt(while(noLargerThan(identifier(num)),
number(10))), stmtList(stmt(if(largerThan(identifier(num)),
number(5))), stmtList(stmt(print(identifier(vOne))))))),
stmtList(stmt(print(identifier(vTwo)))))))))),
stmtList(stmt(assignment(identifier(num)),
plus(identifier(num)), number(1))))))))))),
stmtList(stmt(print(identifier(num)))))))))))))))
```

Intermediate code generator

Data structure: Linked list

Generating: traverse the nodes on the parse tree and put the corresponding intermediate codes in the linked list



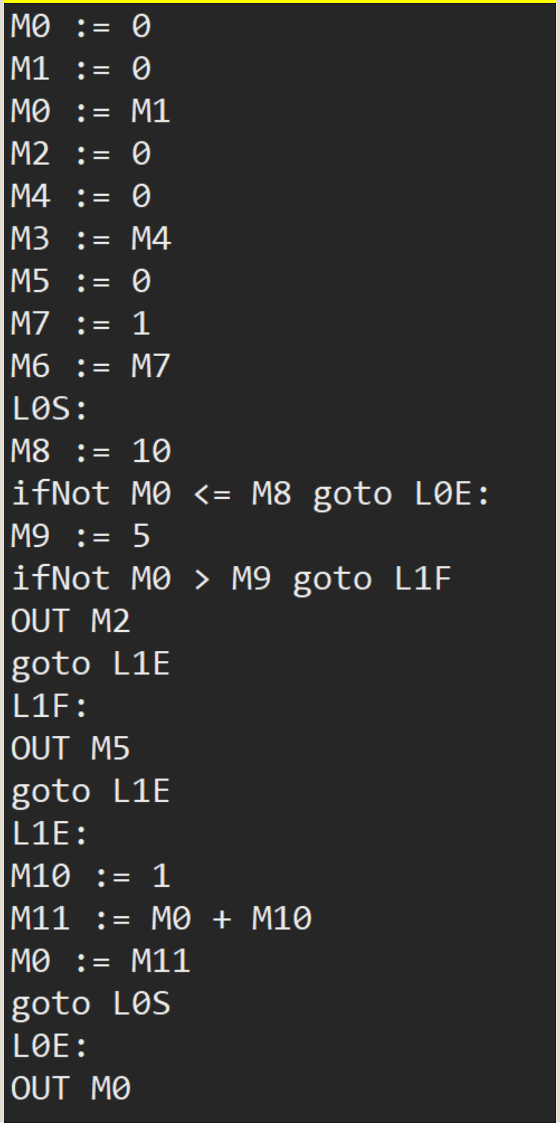
```
public class ICGenerator {
    static int mLocation =0;
    static int counter=0;
    static int labelCounter=0;
    public static Queue<String> ic=new LinkedList<String>();
    public static HashMap<String,String> varList=new HashMap<String,String>();

    public ICGenerator(String s) {
    public ICGenerator(AST t) {

    public static void hProgram(AST t) {

    public static void hStatementList(AST t) {

    public static void hStatement(AST t) {

    public static void hDeclare(AST t,Queue<String> s) {

    public static void hAssignment(AST t,Queue<String> s) {

    public static void hIf(AST t, Queue<String> s) {

    public static void hWhile(AST t, Queue<String> s) {

    public static void hBExpression(AST t, Queue<String> s) {

    public static String hLExpression(AST t,Queue<String> s) {

    public static String hHExpression(AST t,Queue<String> s) {

    public static String hIdentifier(AST t,Queue<String> s) {

    public static String hNumber(AST t, Queue<String> s) {

    public static void hPrint(AST t) {
```

Intermediate code generator

M+index: location in memory
L+index: Label

Operator:
Assignment : ":="
Add : "+"
Minus : "-"
Multiply : "*"
Divide : "/"
Equal : "=="
Not equal : "!="
Larger than : ">"
No larger than : "<="
Less than : "<"
No less than : ">="
Jump to label: "goto"
Print : "OUT"

Intermediate code



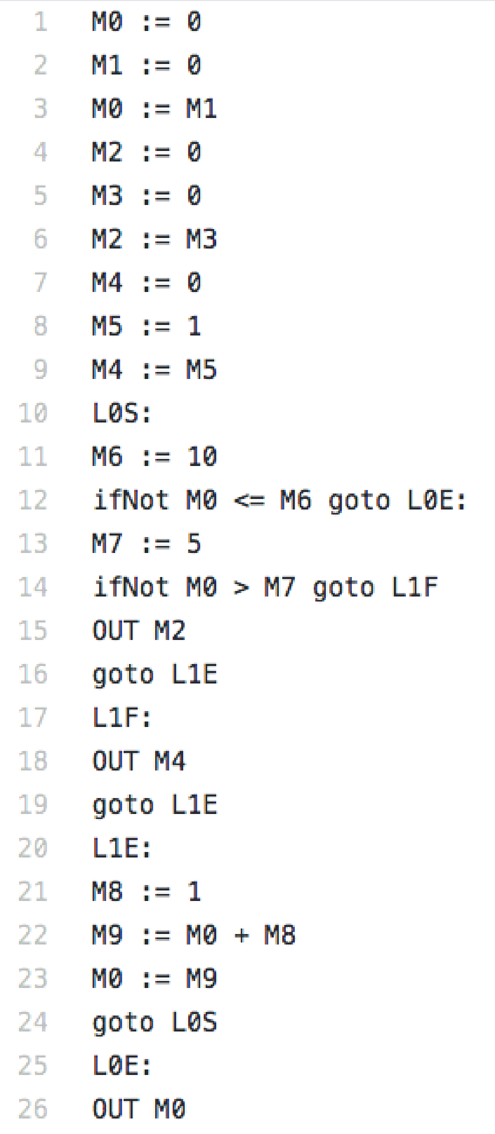
```
M0 := 0
M1 := 0
M0 := M1
M2 := 0
M4 := 0
M3 := M4
M5 := 0
M7 := 1
M6 := M7
L0S:
M8 := 10
ifNot M0 <= M8 goto L0E:
M9 := 5
ifNot M0 > M9 goto L1F
OUT M2
goto L1E
L1F:
OUT M5
goto L1E
L1E:
M10 := 1
M11 := M0 + M10
M0 := M11
goto L0S
L0E:
OUT M0
```

Runtime

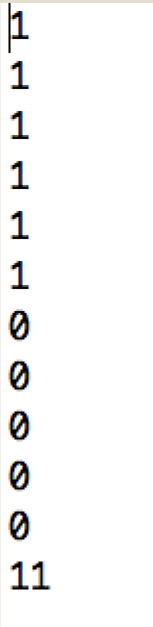
Here are the steps of implementation.

1. Analyze the pattern of the intermediate code on the current line
2. Parse the code
3. Evaluate the code depending on its pattern and save the result into the table
4. Go back to Step 1 if there are still more processes

Runtime

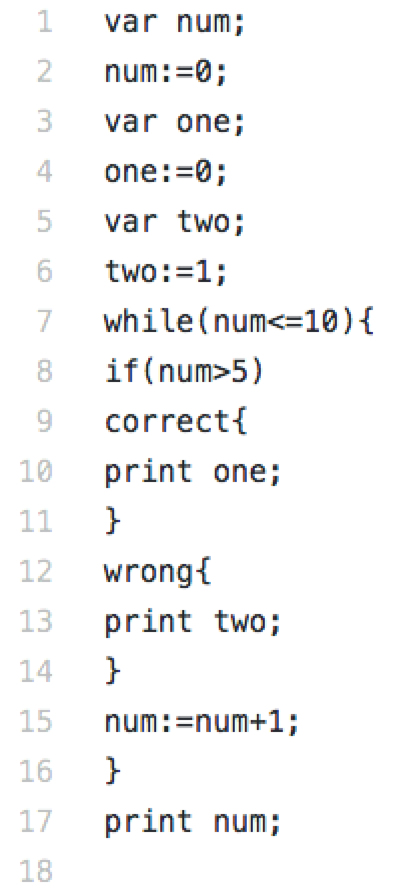


```
M0 := 0
M1 := 0
M0 := M1
M2 := 0
M3 := 0
M2 := M3
M4 := 0
M5 := 1
M4 := M5
L0S:
M6 := 10
ifNot M0 <= M6 goto L0E:
M7 := 5
ifNot M0 > M7 goto L1F
OUT M2
goto L1E
L1F:
OUT M4
goto L1E
L1E:
M8 := 1
M9 := M0 + M8
M0 := M9
goto L0S
L0E:
OUT M0
```



```
1
1
1
1
1
1
0
0
0
0
0
11
```

Source:



```
var num;
num:=0;
var one;
one:=0;
var two;
two:=1;
while(num<=10){
if(num>5)
correct{
print one;
}
wrong{
print two;
}
num:=num+1;
}
print num;
```