SER502-Spring2018-Team2

Zeyong Cai
Zitong Wei
Huichuan Wu
Binbin Yan

Outline

- Language Design
- Grammar
- Intermediate Code
- Compiler Architecture
- Frontend
- Intermediate Code Generator
- Runtime
- Sample

Language Design

- Paradigm: Imperative
- Name: Godfather
- Features
  - Primitives: integer, boolean value
  - Flow control: if-else, while
  - Assignment operator: “=”
  - Arithmetic operators: “+”, “-”, “*”, “/”, “%”
  - Relational operators: “==”, “≠”, “≥”, > ≤ <
  - Supports parentheses to override arithmetic precedence: “(” arithmetic expression “)”
  - A special statement “print” used to output an arithmetic expression to console
- Constrains
  - Do not support function
  - Declarations can only be placed at the top of a program

Grammar

```
program → decls stmts
decls → decl decls_rest | ε
decls_rest →  decl decls_rest | ε
decl → type id ';'
stmts → stmt | stmts_rest | ε
stmts_rest → stmt stmts_rest | ε
stmt → id '=' arith_expr ';'
        | id '=' bool_expr ';'
        | 'if' '(' bool_expr ')' '{' stmts '}'
        | 'if' '(' bool_expr ')' '{' stmts '}' 'else' '{' stmts '}'
        | 'while' '(' bool_expr ')' '{' stmts '}'
        | 'print' '(' arith_expr ')'
bool_expr → arith_expr '==' arith_expr
                | arith_expr '!=' arith_expr
                | arith_expr '>' arith_expr
                | arith_expr '<' arith_expr
                | arith_expr '>=' arith_expr
                | arith_expr '<=' arith_expr
                | bool_value
```

```
arith_expr → term airth_expr_rest
airth_expr_rest → '+' term airth_expr_rest | '-' term airth_expr_rest | ε
term → factor term_rest
term_rest → '*' factor term_rest | '/' factor term_rest | '%' factor term_rest | ε
factor → number | '(' airth_expr ')' | id
type → 'int' | 'bool'
id → [a-z|A-Z]+
num → [0-9]+
bool_value → 'true' | 'false'
```

Intermedia Code —— Three-Address Code

An three-address code has one operator and most three operands. An expression has more than one operator will be translated to multiple instructions.

E.g: y = x + 1 + 2 ----> t1 = x + 1; y = t1 + 2;

In Three-Address Instructions. An address can be a:

- **Name**. usually is an identifier, as a pointer to its symbol table entry.
- **Constant**. can be an integer or boolean value.
- **Compiler-generated temporary**. is a temporary identifier used to save partial result and will be reused or combined later.

Intermedia Code —— Instruction Design

Our intermediate code is in MIPS Style. The operator is on the left side and operands are on the right side.

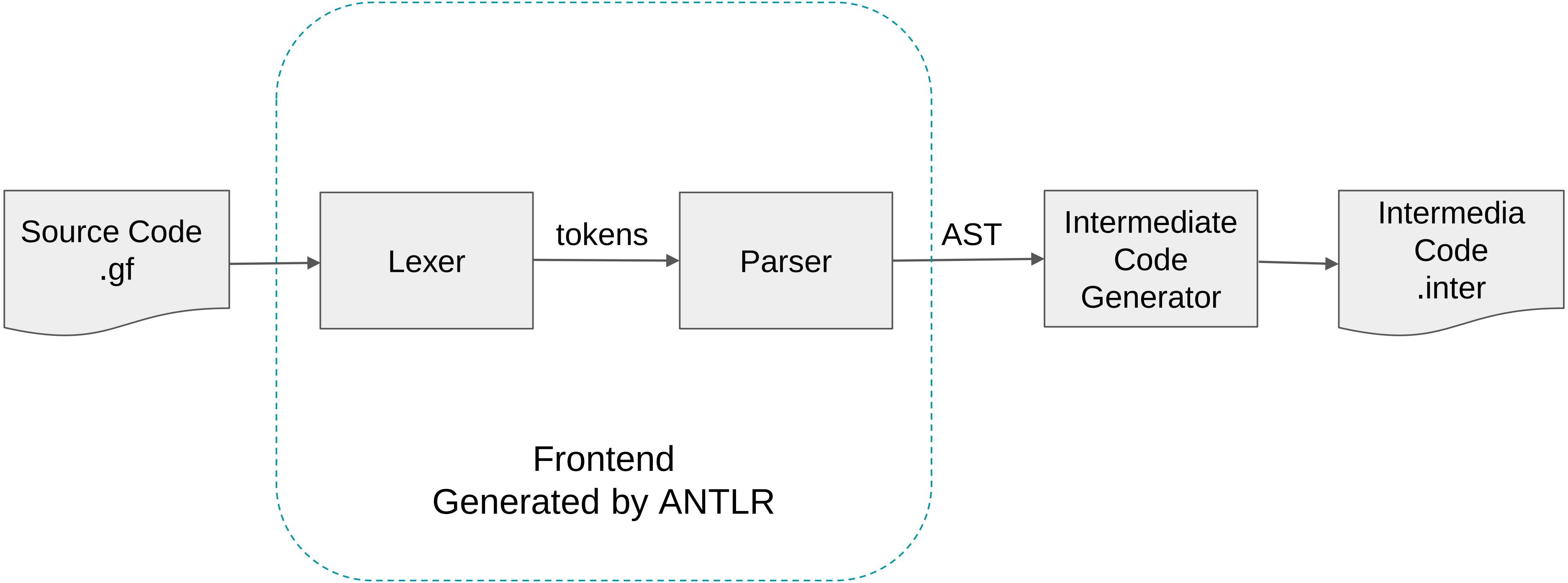
- Assign
  - Rd = Rs ----> move Rd Rs
- Arithmetic
  - Rd = Rs + Rt ----> add Rd Rs Rt
  - Rd = Rs - Rt ----> sub Rd Rs Rt
  - Rd = Rs * Rt ----> mul Rd Rs Rt
  - Rd = Rs / Rt ----> div Rd Rs Rt
  - Rd = Rs % Rt ----> rem Rd Rs Rt

Intermedia Code —— Instruction Design

- Branch
  - Branch to Label iffalse(Rs == Rt) ----> bneq Rs Rt Label
  - Branch to Label iffalse(Rs != Rt) ----> beq Rs Rt Label
  - Branch to Label iffalse(Rs < Rt) ----> bnlt Rs Rt Label
  - Branch to Label iffalse(Rs > Rt) ----> bngt Rs Rt Label
  - Branch to Label iffalse(Rs <= Rt) ----> bnle Rs Rt Label
  - Branch to Label iffalse(Rs >= Rt) ----> bnge Rs Rt Label
- Jump
  - Jump to Label unconditionally ----> j Label
- Extension
  - Print Rs to Console ----> print Rs

An instruction can be marked with an label, E.g: LB1: add rd rs rt .

Compiler Architecture



Source Code .gf

Lexer

tokens

Parser

AST

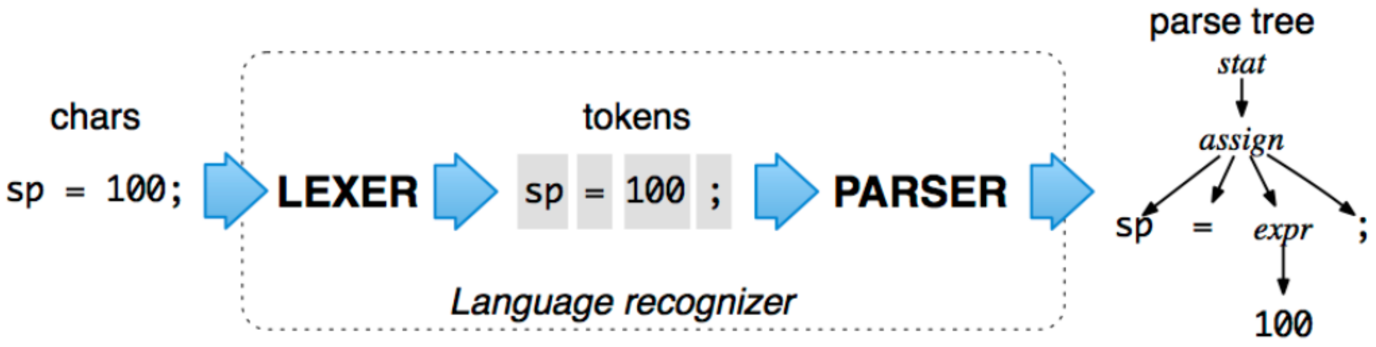
Intermediate Code Generator

Intermedia Code .inter

Frontend
Generated by ANTLR

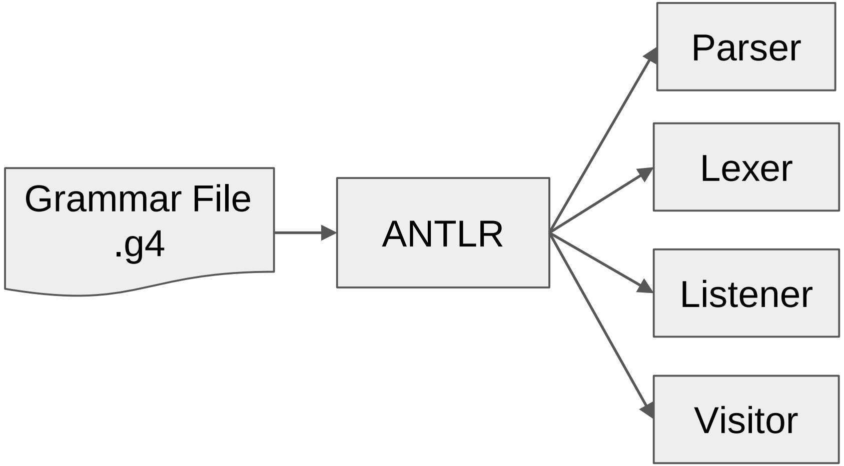
Frontend (Lexer and Parser)

- **Lexer**: sperate a stream of characters into different words, which is the token list.
- **Parser**: generate a parse tree with the token list from the Lexer.
- Antlr makes use of the grammar file we defined to generate a lexer for lexical analysis and a parser for parse tree.



Frontend (Lexer and Parser)

- ANTLR accepts a grammar file to generate the parse and lexer.
- ANTLR also genereates a Listener and Visitor which can be used to traverse the parse tree.



- **Listener**: is called by the ANTLR-provided walker object, it cannot be controlled by us.
- **Visitor**: walk their children with explicit visit calls and we can control its path. Developers can define a return type for visitor to implement more complex manipulations.

Frontend —— Grammar File for ANTLR

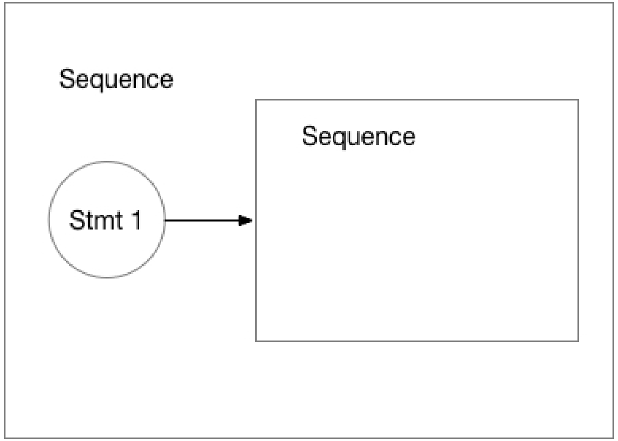
```
grammar GodFather;
program : decls stmts;
decls : (decl)*;
decl : type=('int' | 'bool') ID ';';
stmts : (stmt)*;
stmt : ID '=' arith_expr ';' # stmtArithAssign
      | ID '=' bool_expr ';' # stmtBoolAssign
      | 'if' '(' bool_expr ')' '{' stmts '}' # stmtIf
      | 'if' '(' bool_expr ')' '{' stmts '}' 'else' '{' stmts '}' # stmtIfElse
      | 'while' '(' bool_expr ')' '{' stmts '}' # stmtWhile
      | 'print' '(' arith_expr ')' ';'  # stmtPrint
      ;
bool_expr : arith_expr op=('==' | '!=' | '>' | '<' | '>=' | '<=') arith_expr #boolExprCmp
             | value=('true' | 'false') #boolExprValue
             ;
arith_expr : term (op=('+' | '-') term)*;
term : factor (op=('*' | '/' | '%') factor)*;
factor : NUMBER | '(' arith_expr ')' | ID;
```

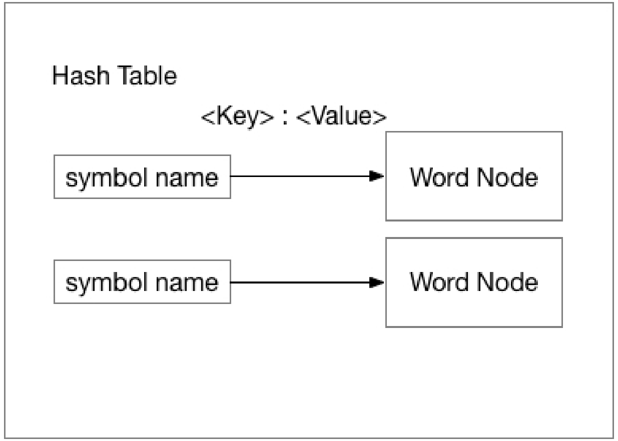
Frontend —— Grammar File for ANTLR

```
LE : '<=';
SEMI : ';';
OR : '||';
GT : '>';
ASSIGN : '=';
GE : '>=';
EQ : '==';
PLUS : '+';
MINUS : '-';
NE : '!=';
MUL : '*';
LT : '<';
DIV : '/';
INT : 'int';
BOOL : 'bool';
NUMBER : [0-9]+ ;
TRUE : 'true';
FALSE: 'false';
ID : [a-z|A-Z]+ ;
WS : [ \t\r\n]+ -> skip ;
```

Intermedia Code Generator —— Data Structure

LinkedList: used to express the execution order.
Hash Table: used to save global declared id as a symbol table.



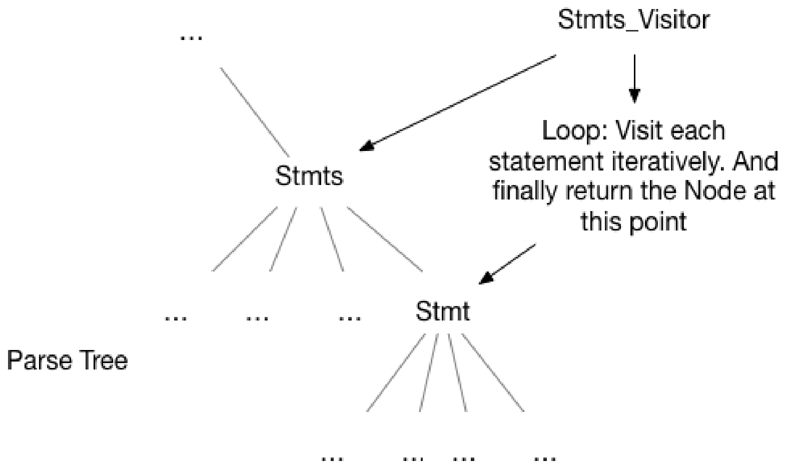


Intermedia Code Generator —— Translation Process

Intermediate code translation is based on the ANTLR Visitor interfaces.
There are two steps for translation.

- Step 1: Traverse

Use Visitor to access each node on the parse tree and generate corresponding Node at this point and return it to higher-level.


Stmts_Visitor
...
Loop: Visit each statement iteratively. And finally return the Node at this point
Stmts
...
...
...
Stmt
Parse Tree
...
...
...
...

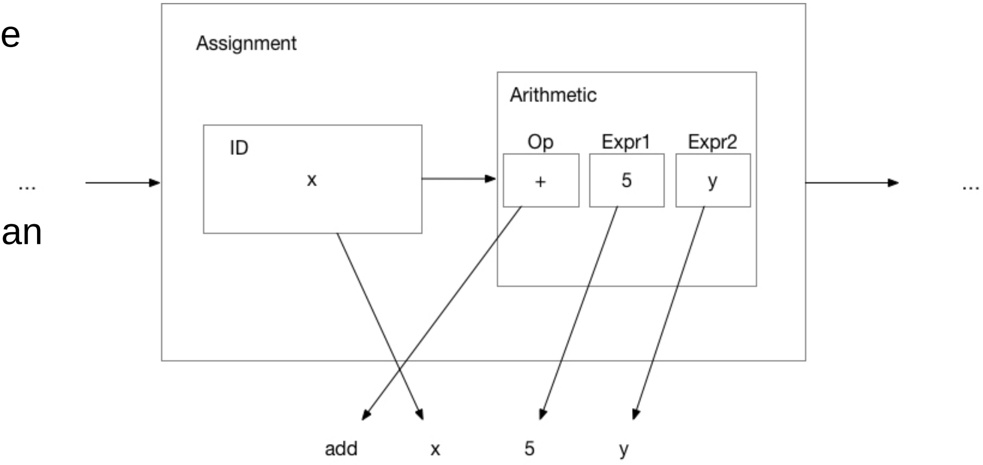
Intermedia Code Generator —— Translation Process

- Step 2: Print

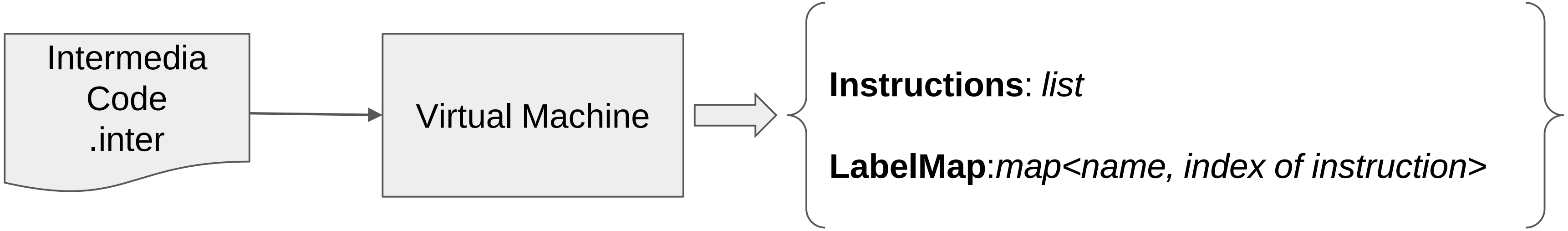
After Step 1, we have will have a generated linked list to store all instructions. Then, we will generate intermediate recursively for each node.

When we try to generate intermediate code for Assignment node. Since there is an id node and an arithmetic node inside, we go deep in those two nodes and generate intermediate code for them, firstly.

Finally, A file with ‘.inter’ file will be created at the same location as provided ‘.gf’ file.



Runtime —— Parsing Phase

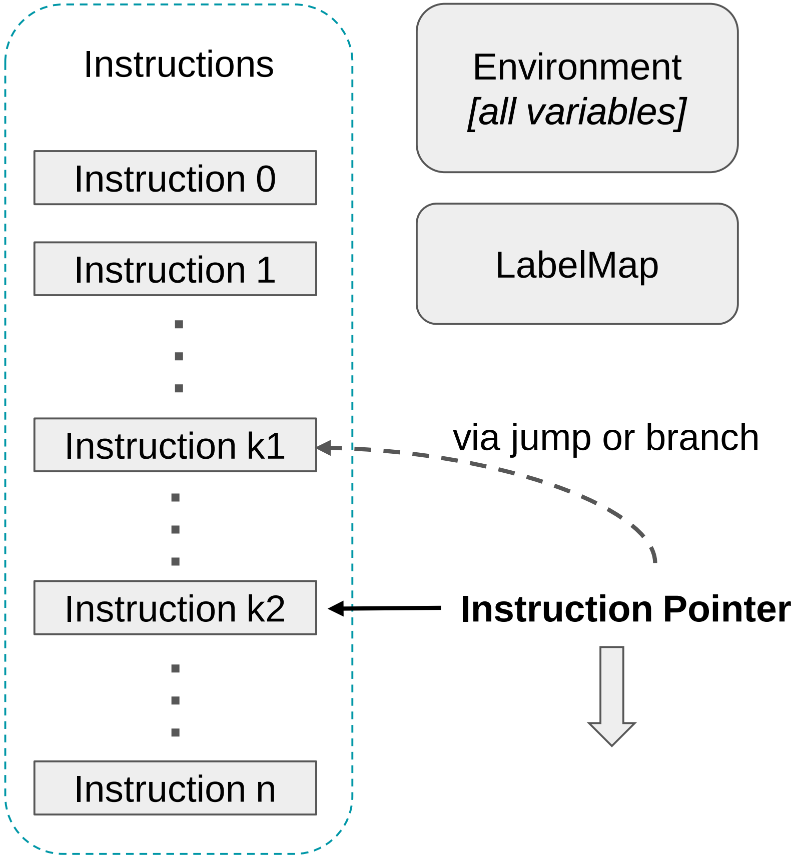


Instructions: *list*

LabelMap:*map<name, index of instruction>*

Each line in the intermediate code file will be parsed to an instruction.
The LabelMap is used to store the mapping relationship between a label and the index of an instruction.

Runtime —— Execution Phase



Core Data Structures

- **Instruction Pointer**: *int.* Used to indicate the next instruction.
- **Environment**:*map<string, integer>.* Used to store all the variables at runtime. All the variables will be updated and queried from the environment. There're no mutiple environments because we do not support nested declarations.

Running Process

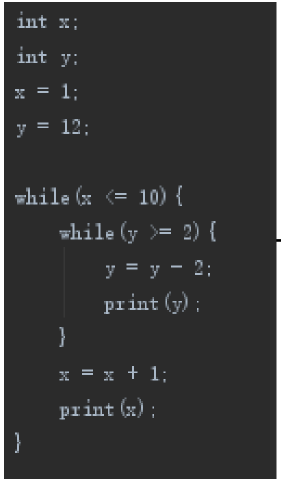
- Instructions are executed one by one from the very first instruction.
- A program completes when the Instruction Pointer points the n + 1 instruction.
- The Instruction Pointer can be changed by an jump or branch instruction.

Sample

Four Samples to show the accuracy of language:

- While loop sample
- If statement sample
- Compound expression sample
- Factorial sample

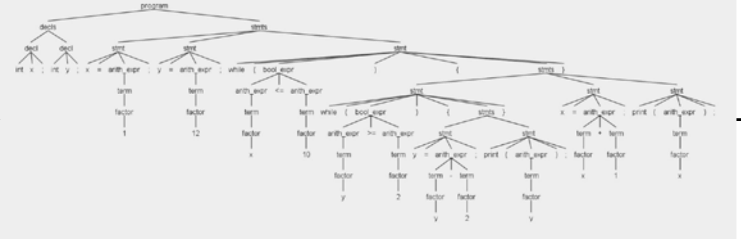
While Loop Sample



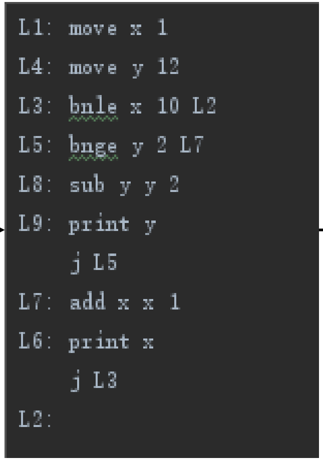
```
int x;
int y;
x = 1;
y = 12;

while(x <= 10){
    while(y >= 2){
        y = y - 2;
        print(y);
    }
    x = x + 1;
    print(x);
}
```

Source Code

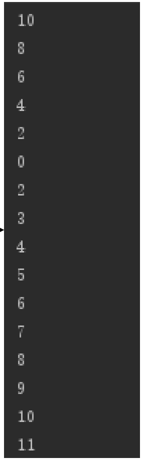


Praser Tree



```
L1: move x 1
L4: move y 12
L3: bnle x 10 L2
L5: bnge y 2 L7
L8: sub y y 2
L9: print y
    j L5
L7: add x x 1
L6: print x
    j L3
L2:
```

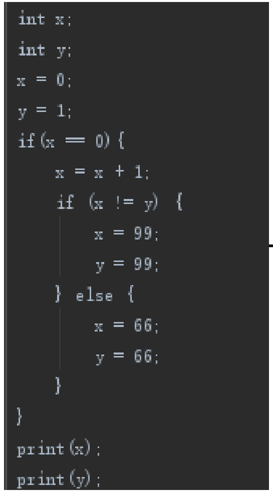
Intermediate Code



```
10
8
6
4
2
0
2
3
4
5
6
7
8
9
10
11
```

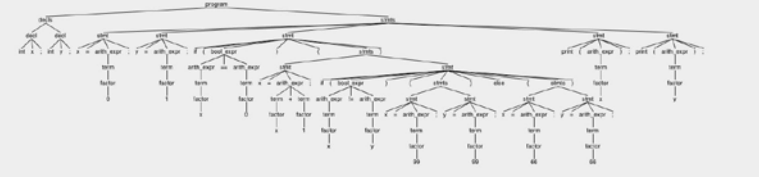
Result

If Statement Sample

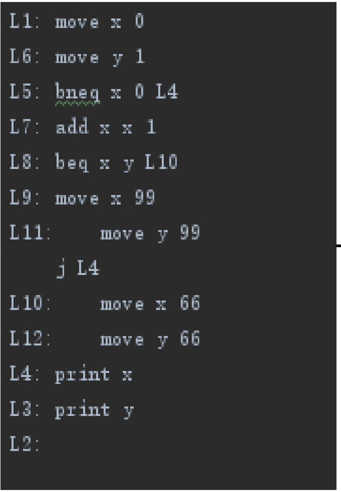


```
int x;
int y;
x = 0;
y = 1;
if(x == 0){
    x = x + 1;
    if (x != y) {
        x = 99;
        y = 99;
    } else {
        x = 66;
        y = 66;
    }
}
print(x);
print(y);
```

Source Code

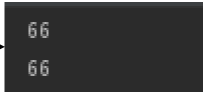


Praser Tree



```
L1:  move x 0
L6:  move y 1
L5:  bneq x 0 L4
L7:  add x x 1
L8:  beq x y L10
L9:  move x 99
L11:    move y 99
     j L4
L10:    move x 66
L12:    move y 66
L4:  print x
L3:  print y
L2:
```

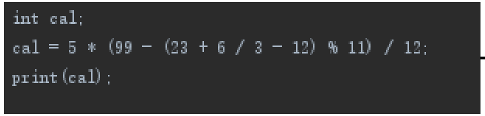
Intermediate Code



```
66
66
```

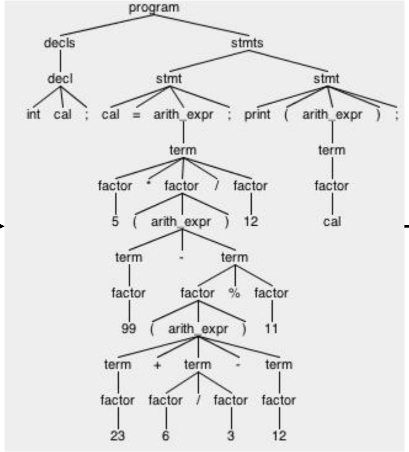
Result

Compound Expression Sample

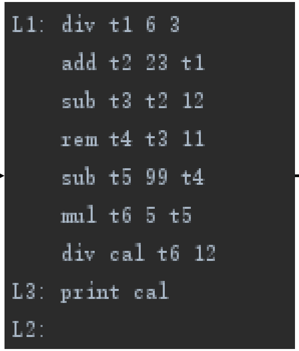


```
int cal;
cal = 5 * (99 - (23 + 6 / 3 - 12) % 11) / 12;
print(cal);
```

Source Code

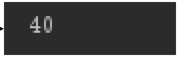


Praser Tree



```
L1: div t1 6 3
    add t2 23 t1
    sub t3 t2 12
    rem t4 t3 11
    sub t5 99 t4
    mul t6 5 t5
    div cal t6 12
L3: print cal
L2:
```

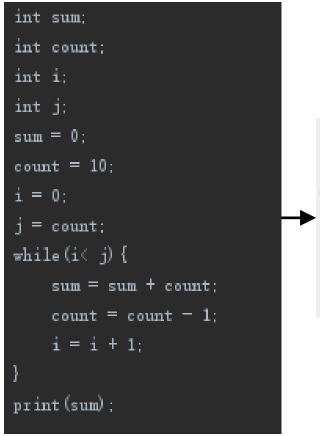
Intermediate Code



```
40
```

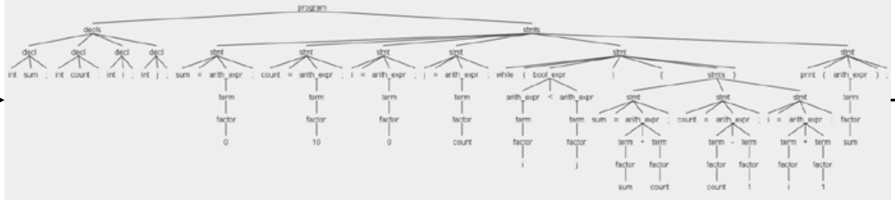
Result

Factorial Sample

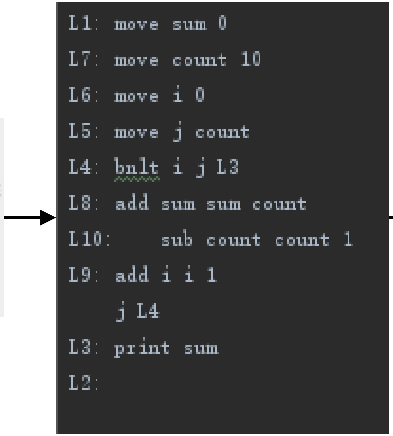


```
int sum;
int count;
int i;
int j;
sum = 0;
count = 10;
i = 0;
j = count;
while(i< j){
    sum = sum + count;
    count = count - 1;
    i = i + 1;
}
print(sum);
```

Source Code

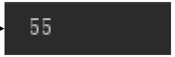

program
decls
stmts
decl decl decl decl
stmt stmt stmt stmt stmt stmt
int sum ; int count ; int i ; int j ;
sum = arith_expr ; count = arith_expr ; i = arith_expr ; j = arith_expr ;
while (bool_expr) { stmts }
print (arith_expr) ;
term term term term
factor factor factor factor
0 10 0 count
arith_expr < arith_expr
stmt stmt stmt
term term
factor factor
i j
sum = arith_expr ; count = arith_expr ; i = arith_expr ;
term + term term - term term + term
factor factor factor factor factor factor
sum count count 1 i 1
term
factor
sum

Praser Tree



```
L1: move sum 0
L7: move count 10
L6: move i 0
L5: move j count
L4: bnlt i j L3
L8: add sum sum count
L10:    sub count count 1
L9: add i i 1
    j L4
L3: print sum
L2:
```

Intermediate Code



```
55
```

Result