|  |  |  |  |
| --- | --- | --- | --- |
| فاز اول پروژه ی درس کامپایلر  استاد درس: دکتر رزازی  زهرا نژادیان 9433105- سپیده ملانوروزی 9431072 | |  |  |
| **تاریخ تحویل : دوشنبه 20 آبان – موعد تحویل : دوشنبه 20 ابان** | | | |

**Input :**

class Program{

int var, integerId = 4;

static void \_main(){

var = -23 - 22;

print("N

Sanity.");

}

// Comment.

bool boolean(){

if(var < 8 && 32 - var > var\_2){

var = var | (integerId - 2);

var = var & 0b10010010;

while(true){

var = 3;

break;

}

}elseif( var > 8 ){

var = var & 0b10010010;

}

else var = var & 0b10010010;

}

real \_func(int i, int j){

return i % j \* 3.52;

}

}

**Output :**

macros ->/\* Lambda \*/

classes ->/\* Lambda \*/

symbol\_decs ->/\* Lambda \*/

return\_type -> INT\_TYPE

var\_type -> return\_type

var\_list\_item -> ID

var\_list -> var\_list\_item

exp -> INTEGER

item1 -> ID ASSIGNMENT exp

var\_list\_item -> item1

var\_list -> var\_list COMMA var\_list\_item

var\_dec -> var\_type var\_list SEMICOLON

symbol\_dec -> var\_dec

symbol\_decs -> symbol\_decs symbol\_dec

formal\_arguments ->/\* Lambda \*/

statements\_list ->/\* Lambda \*/

lvalue1 -> ID

lvalue -> lvalue1

exp -> INTEGER

unary\_operation -> SUBTRACTION exp

exp -> unary\_operation

exp -> INTEGER

binary\_operation -> exp SUBTRACTION exp

exp -> binary\_operation

assignment -> lvalue ASSIGNMENT exp SEMICOLON

statement -> assignment

statements\_list -> statements\_list statement

print -> PRINT LP STRING RP SEMICOLON

statement -> print

statements\_list -> statements\_list statement

block -> LCB statements\_list RCB

func\_body -> ID LP formal\_arguments RP block

func\_dec -> STATIC VOID func\_body

symbol\_dec -> func\_dec

symbol\_decs -> symbol\_decs symbol\_dec

THIS IS SYMBOL TABLE

Program 0

var 1

integerId 2

\_main 3

boolean 4

var\_2 5

\_func 6

i 7

j 8

exp -> bitwise\_operation

assignment -> lvalue ASSIGNMENT exp SEMICOLON

statement -> assignment

statements\_list -> statements\_list statement

lvalue1 -> ID

lvalue -> lvalue1

lvalue1 -> ID

lvalue -> lvalue1

exp -> lvalue

exp -> INTEGER

bitwise\_operation -> exp BITWISE\_AND exp

exp -> bitwise\_operation

assignment -> lvalue ASSIGNMENT exp SEMICOLON

statement -> assignment

statements\_list -> statements\_list statement

exp -> TRUE

statements\_list ->/\* Lambda \*/

lvalue1 -> ID

lvalue -> lvalue1

exp -> INTEGER

assignment -> lvalue ASSIGNMENT exp SEMICOLON

statement -> assignment

statements\_list -> statements\_list statement

break -> BREAK SEMICOLON

statement -> break

statements\_list -> statements\_list statement

block -> LCB statements\_list RCB

while -> WHILE LP exp RP block

statement -> while

statements\_list -> statements\_list statement

block -> LCB statements\_list RCB

lvalue1 -> ID

lvalue -> lvalue1

exp -> lvalue

exp -> INTEGER

comparison\_operation -> exp GT exp

exp -> comparison\_operation

return\_type -> BOOL\_TYPE

var\_type -> return\_type

formal\_arguments ->/\* Lambda \*/

statements\_list ->/\* Lambda \*/

lvalue1 -> ID

lvalue -> lvalue1

exp -> lvalue

exp -> INTEGER

comparison\_operation -> exp LT exp

exp -> comparison\_operation

exp -> INTEGER

lvalue1 -> ID

lvalue -> lvalue1

exp -> lvalue

binary\_operation -> exp SUBTRACTION exp

exp -> binary\_operation

lvalue1 -> ID

lvalue -> lvalue1

exp -> lvalue

comparison\_operation -> exp GT exp

exp -> comparison\_operation

logical\_operation -> exp AND exp

exp -> logical\_operation

statements\_list ->/\* Lambda \*/

lvalue1 -> ID

lvalue -> lvalue1

lvalue1 -> ID

lvalue -> lvalue1

exp -> lvalue

lvalue1 -> ID

lvalue -> lvalue1

exp -> lvalue

exp -> INTEGER

binary\_operation -> exp SUBTRACTION exp

exp -> binary\_operation

exp -> LP exp RP

bitwise\_operation -> exp BITWISE\_OR exp

statements\_list ->/\* Lambda \*/

lvalue1 -> ID

lvalue -> lvalue1

lvalue1 -> ID

lvalue -> lvalue1

exp -> lvalue

exp -> INTEGER

bitwise\_operation -> exp BITWISE\_AND exp

exp -> bitwise\_operation

assignment -> lvalue ASSIGNMENT exp SEMICOLON

statement -> assignment

statements\_list -> statements\_list statement

block -> LCB statements\_list RCB

elseif -> ELSEIF LP exp RP block

elseifs -> elseif

lvalue1 -> ID

lvalue -> lvalue1

lvalue1 -> ID

lvalue -> lvalue1

exp -> lvalue

exp -> INTEGER

bitwise\_operation -> exp BITWISE\_AND exp

exp -> bitwise\_operation

assignment -> lvalue ASSIGNMENT exp SEMICOLON

statement -> assignment

block -> statement

if -> IF LP exp RP block elseifs ELSE block

statement -> if

statements\_list -> statements\_list statement

block -> LCB statements\_list RCB

func\_body -> ID LP formal\_arguments RP block

func\_dec -> var\_type func\_body

symbol\_dec -> func\_dec

symbol\_decs -> symbol\_decs symbol\_dec

return\_type -> REAL\_TYPE

var\_type -> return\_type

return\_type -> INT\_TYPE

formal\_argument -> return\_type ID

formal\_arguments\_list -> formal\_argument

return\_type -> INT\_TYPE

formal\_argument -> return\_type ID

formal\_arguments\_list -> formal\_arguments\_list COMMA formal\_argument

formal\_arguments -> formal\_arguments\_list

statements\_list ->/\* Lambda \*/

lvalue1 -> ID

lvalue -> lvalue1

exp -> lvalue

lvalue1 -> ID

lvalue -> lvalue1

exp -> lvalue

binary\_operation -> exp MODULO exp

exp -> binary\_operation

exp -> REAL

binary\_operation -> exp MULTIPLICATION exp

exp -> binary\_operation

return -> RETURN exp SEMICOLON

statement -> return

statements\_list -> statements\_list statement

block -> LCB statements\_list RCB

func\_body -> ID LP formal\_arguments RP block

func\_dec -> var\_type func\_body

symbol\_dec -> func\_dec

symbol\_decs -> symbol\_decs symbol\_dec

class -> CLASS ID LCB symbol\_decs RCB

classes -> classes class

program -> macros classes

**Code :**

import ply.yacc as yacc

import classes.lexer as l

class Parser:

tokens = l.Lexer().tokens

def p\_program(self, p):

'''program : macros classes'''

print('''program -> macros classes''')

def p\_macros(self, p):

"""macros : macros macro"""

print("""macros -> macros macro""")

def p\_macros\_e(self, p):

"""macros : """

print("""macros ->/\* Lambda \*/""")

def p\_macro(self, p):

"""macro : reference"""

print("""macro -> reference""")

def p\_reference(self, p):

"""reference : REFERENCE STRING"""

print("""reference -> REFERENCE STRING""")

def p\_classes(self, p):

"""classes : classes class"""

print("""classes -> classes class""")

def p\_classes\_e(self, p):

"""classes : """

print("""classes ->/\* Lambda \*/""")

def p\_class(self, p):

"""class : CLASS ID LCB symbol\_decs RCB"""

print("""class -> CLASS ID LCB symbol\_decs RCB""")

def p\_symbol\_decs(self, p):

"""symbol\_decs : symbol\_decs symbol\_dec"""

print("""symbol\_decs -> symbol\_decs symbol\_dec""")

def p\_symbol\_decs\_e(self, p):

"""symbol\_decs : """

print("""symbol\_decs ->/\* Lambda \*/""")

def p\_symbol\_dec\_1(self, p):

"""symbol\_dec : var\_dec"""

print("""symbol\_dec -> var\_dec""")

def p\_symbol\_dec\_2(self, p):

"""symbol\_dec : func\_dec"""

print("""symbol\_dec -> func\_dec""")

def p\_var\_dec(self, p):

"""var\_dec : var\_type var\_list SEMICOLON"""

print("""var\_dec -> var\_type var\_list SEMICOLON""")

def p\_var\_type\_1(self, p):

"""var\_type : return\_type"""

print("""var\_type -> return\_type""")

def p\_var\_type\_1\_1(self, p):

"""var\_type : lvalue1"""

print("""var\_type -> lvalue1""")

def p\_var\_type\_2(self, p):

"""var\_type : STATIC return\_type"""

print("""var\_type -> STATIC return\_type""")

def p\_var\_type\_2\_1(self, p):

"""var\_type : STATIC lvalue1"""

print("""var\_type -> STATIC lvalue1""")

def p\_return\_type\_1(self, p):

"""return\_type : INT\_TYPE"""

print("""return\_type -> INT\_TYPE""")

def p\_return\_type\_2(self, p):

"""return\_type : REAL\_TYPE"""

print("""return\_type -> REAL\_TYPE""")

def p\_return\_type\_3(self, p):

"""return\_type : BOOL\_TYPE"""

print("""return\_type -> BOOL\_TYPE""")

def p\_return\_type\_4(self, p):

"""return\_type : STRING\_TYPE"""

print("""return\_type -> STRING\_TYPE""")

def p\_var\_list\_1(self, p):

"""var\_list : var\_list COMMA var\_list\_item"""

print("""var\_list -> var\_list COMMA var\_list\_item""")

def p\_var\_list\_2(self, p):

"""var\_list : var\_list\_item"""

print("""var\_list -> var\_list\_item""")

def p\_item1(self, p):

"""item1 : ID ASSIGNMENT exp"""

print("""item1 -> ID ASSIGNMENT exp""")

def p\_var\_list\_item\_2(self, p):

"""var\_list\_item : item1"""

print("""var\_list\_item -> item1""")

def p\_var\_list\_item\_1(self, p):

"""var\_list\_item : ID"""

print("""var\_list\_item -> ID""")

def p\_func\_dec(self, p):

"""func\_dec : var\_type func\_body"""

print("""func\_dec -> var\_type func\_body""")

def p\_func\_dec\_1(self, p):

"""func\_dec : VOID func\_body"""

print("""func\_dec -> VOID func\_body""")

def p\_func\_dec\_2(self, p):

"""func\_dec : STATIC VOID func\_body"""

print("""func\_dec -> STATIC VOID func\_body""")

def p\_func\_body(self, p):

"""func\_body : ID LP formal\_arguments RP block"""

print("""func\_body -> ID LP formal\_arguments RP block""")

def p\_formal\_arguments(self, p):

"""formal\_arguments : formal\_arguments\_list"""

print("""formal\_arguments -> formal\_arguments\_list""")

def p\_formal\_arguments\_e(self, p):

"""formal\_arguments : """

print("""formal\_arguments ->/\* Lambda \*/""")

def p\_formal\_arguments\_list(self, p):

"""formal\_arguments\_list : formal\_arguments\_list COMMA formal\_argument"""

print("""formal\_arguments\_list -> formal\_arguments\_list COMMA formal\_argument""")

def p\_formal\_arguments\_list\_1(self, p):

"""formal\_arguments\_list : formal\_argument"""

print("""formal\_arguments\_list -> formal\_argument""")

def p\_formal\_argument(self, p):

"""formal\_argument : return\_type ID"""

print("""formal\_argument -> return\_type ID""")

def p\_formal\_argument\_1(self, p):

"""formal\_argument : lvalue1 ID"""

print("""formal\_argument -> lvalue1 ID""")

def p\_block(self, p):

"""block : LCB statements\_list RCB"""

print("""block -> LCB statements\_list RCB""")

def p\_block\_s(self, p):

"""block : statement"""

print("""block -> statement""")

def p\_statements\_list(self, p):

"""statements\_list : statements\_list statement"""

print("""statements\_list -> statements\_list statement""")

def p\_statements\_list\_e(self, p):

"""statements\_list : """

print("""statements\_list ->/\* Lambda \*/""")

def p\_statement(self, p):

"""statement : SEMICOLON"""

print("""statement -> SEMICOLON""")

def p\_statement0(self, p):

"""statement : exp SEMICOLON"""

print("""statement -> exp""")

def p\_statement\_1(self, p):

"""statement : assignment"""

print("""statement -> assignment""")

def p\_statement\_2(self, p):

"""statement : print"""

print("""statement -> print""")

def p\_statement\_3(self, p):

"""statement : statement\_var\_dec"""

print("""statement -> statement\_var\_dec""")

def p\_statement\_4(self, p):

"""statement : if"""

print("""statement -> if""")

def p\_statement\_5(self, p):

"""statement : for"""

print("""statement -> for""")

def p\_statement\_6(self, p):

"""statement : while"""

print("""statement -> while""")

def p\_statement\_7(self, p):

"""statement : return"""

print("""statement -> return""")

def p\_statement\_8(self, p):

"""statement : break"""

print("""statement -> break""")

def p\_statement\_9(self, p):

"""statement : continue"""

print("""statement -> continue""")

def p\_assignment(self, p):

"""assignment : lvalue ASSIGNMENT exp SEMICOLON"""

print("""assignment -> lvalue ASSIGNMENT exp SEMICOLON""")

def p\_lvalue\_1(self, p):

"""lvalue : lvalue1 %prec LVALI"""

print("""lvalue -> lvalue1""")

def p\_lvalue\_2(self, p):

"""lvalue : lvalue2 %prec LVAL"""

print("""lvalue -> lvalue2""")

def p\_lval2(self, p):

"""lvalue2 : ID DOT ID"""

print("""lvalue2 -> ID DOT ID""")

def p\_lval1(self, p):

"""lvalue1 : ID"""

print("""lvalue1 -> ID""")

def p\_print(self, p):

"""print : PRINT LP STRING RP SEMICOLON"""

print("""print -> PRINT LP STRING RP SEMICOLON""")

def p\_statement\_var\_dec(self, p):

"""statement\_var\_dec : return\_type var\_list SEMICOLON"""

print("""statement\_var\_dec -> return\_type var\_list SEMICOLON""")

def p\_statement\_var\_dec\_1(self, p):

"""statement\_var\_dec : lvalue1 var\_list SEMICOLON"""

print("""statement\_var\_dec -> lvalue1 var\_list SEMICOLON""")

def p\_if\_1(self, p):

"""if : IF LP exp RP block %prec IF"""

print("""if -> IF LP exp RP block""")

def p\_if\_2(self, p):

"""if : IF LP exp RP block ELSE block %prec ELSE"""

print("""if -> IF LP exp RP block ELSE block""")

def p\_if\_3(self, p):

"""if : IF LP exp RP block elseifs %prec ELSEIF"""

print("""if -> IF LP exp RP block elseifs""")

def p\_if\_4(self, p):

"""if : IF LP exp RP block elseifs ELSE block %prec ELSEIF"""

print("""if -> IF LP exp RP block elseifs ELSE block""")

def p\_elseifs\_1(self, p):

"""elseifs : elseifs elseif"""

print("""elseifs -> elseifs elseif""")

def p\_elseifs\_2(self, p):

"""elseifs : elseif"""

print("""elseifs -> elseif""")

def p\_elseif(self, p):

"""elseif : ELSEIF LP exp RP block"""

print("""elseif -> ELSEIF LP exp RP block""")

def p\_for(self, p):

"""for : FOR LP ID IN exp TO exp STEPS exp RP block"""

print("""for -> FOR LP ID IN exp TO exp STEPS exp RP block""")

def p\_while(self, p):

"""while : WHILE LP exp RP block"""

print("""while -> WHILE LP exp RP block""")

def p\_return(self, p):

"""return : RETURN exp SEMICOLON"""

print("""return -> RETURN exp SEMICOLON""")

def p\_break(self, p):

"""break : BREAK SEMICOLON"""

print("""break -> BREAK SEMICOLON""")

def p\_continue(self, p):

"""continue : CONTINUE SEMICOLON"""

print("""continue -> CONTINUE SEMICOLON""")

def p\_exp(self, p):

"""exp : INTEGER"""

print("""exp -> INTEGER""")

def p\_exp\_1(self, p):

"""exp : REAL"""

print("""exp -> REAL""")

def p\_exp\_2(self, p):

"""exp : TRUE"""

print("""exp -> TRUE""")

def p\_exp\_3(self, p):

"""exp : FALSE"""

print("""exp -> FALSE""")

def p\_exp\_4(self, p):

"""exp : STRING"""

print("""exp -> STRING""")

def p\_exp\_5(self, p):

"""exp : lvalue"""

print("""exp -> lvalue""")

def p\_exp\_6(self, p):

"""exp : binary\_operation %prec BIOP"""

print("""exp -> binary\_operation""")

def p\_exp\_7(self, p):

"""exp : logical\_operation"""

print("""exp -> logical\_operation""")

def p\_exp\_8(self, p):

"""exp : comparison\_operation %prec COMOP"""

print("""exp -> comparison\_operation""")

def p\_exp\_9(self, p):

"""exp : bitwise\_operation %prec BITOP"""

print("""exp -> bitwise\_operation""")

def p\_exp\_10(self, p):

"""exp : unary\_operation"""

print("""exp -> unary\_operation""")

def p\_exp\_11(self, p):

"""exp : LP exp RP"""

print("""exp -> LP exp RP""")

def p\_exp\_12(self, p):

"""exp : function\_call"""

print("""exp -> function\_call""")

def p\_binary\_operation(self, p):

"""binary\_operation : exp ADDITION exp """

print("""binary\_operation -> exp ADDITION exp """)

def p\_binary\_operation\_1(self, p):

"""binary\_operation : exp SUBTRACTION exp"""

print("""binary\_operation -> exp SUBTRACTION exp""")

def p\_binary\_operation\_2(self, p):

"""binary\_operation : exp MULTIPLICATION exp"""

print("""binary\_operation -> exp MULTIPLICATION exp""")

def p\_binary\_operation\_3(self, p):

"""binary\_operation : exp DIVISION exp"""

print("""binary\_operation -> exp DIVISION exp""")

def p\_binary\_operation\_4(self, p):

"""binary\_operation : exp MODULO exp"""

print("""binary\_operation -> exp MODULO exp""")

def p\_binary\_operation\_5(self, p):

"""binary\_operation : exp POWER exp"""

print("""binary\_operation -> exp POWER exp""")

def p\_binary\_operation\_6(self, p):

"""binary\_operation : exp SHIFT\_LEFT exp"""

print("""binary\_operation -> exp SHIFT\_LEFT exp""")

def p\_binary\_operation\_7(self, p):

"""binary\_operation : exp SHIFT\_RIGHT exp"""

print("""binary\_operation -> exp SHIFT\_RIGHT exp""")

def p\_logical\_operation(self, p):

"""logical\_operation : exp AND exp"""

print("""logical\_operation -> exp AND exp""")

def p\_logical\_operation\_1(self, p):

"""logical\_operation : exp OR exp"""

print("""logical\_operation -> exp OR exp""")

def p\_comparison\_operation\_1(self, p):

"""comparison\_operation : exp LT exp"""

print("""comparison\_operation -> exp LT exp""")

def p\_comparison\_operation\_2(self, p):

"""comparison\_operation : exp LE exp"""

print("""comparison\_operation -> exp LE exp""")

def p\_comparison\_operation\_3(self, p):

"""comparison\_operation : exp GT exp"""

print("""comparison\_operation -> exp GT exp""")

def p\_comparison\_operation\_4(self, p):

"""comparison\_operation : exp GE exp"""

print("""comparison\_operation -> exp GE exp""")

def p\_comparison\_operation\_5(self, p):

"""comparison\_operation : exp EQ exp"""

print("""comparison\_operation -> exp EQ exp""")

def p\_comparison\_operation\_6(self, p):

"""comparison\_operation : exp NE exp"""

print("""comparison\_operation -> exp NE exp""")

def p\_bitwise\_operation\_1(self, p):

"""bitwise\_operation : exp BITWISE\_AND exp"""

print("""bitwise\_operation -> exp BITWISE\_AND exp""")

def p\_bitwise\_operation\_2(self, p):

"""bitwise\_operation : exp BITWISE\_OR exp"""

print("""bitwise\_operation -> exp BITWISE\_OR exp""")

def p\_unary\_operation\_1(self, p):

"""unary\_operation : SUBTRACTION exp %prec UMINUS"""

print("""unary\_operation -> SUBTRACTION exp""")

def p\_unary\_operation\_2(self, p):

"""unary\_operation : NOT exp"""

print("""unary\_operation -> NOT exp""")

def p\_unary\_operation\_3(self, p):

"""unary\_operation : BITWISE\_NOT exp"""

print("""unary\_operation -> BITWISE\_NOT exp""")

def p\_function\_call\_2(self, p):

"""function\_call : lvalue2 function\_call\_body"""

print("""function\_call -> lvalue2 function\_call\_body""")

def p\_function\_call\_1(self, p):

"""function\_call : lvalue1 function\_call\_body"""

print("""function\_call -> lvalue1 function\_call\_body""")

def p\_function\_call\_body(self, p):

"""function\_call\_body : LP actual\_arguments RP"""

print("""function\_call\_body -> LP actual\_arguments RP""")

def p\_actual\_arguments(self, p):

"""actual\_arguments : actual\_arguments\_list"""

print("""actual\_arguments -> actual\_arguments\_list""")

def p\_actual\_arguments\_e(self, p):

"""actual\_arguments : """

print("""actual\_arguments ->/\* Lambda \*/""")

def p\_actual\_arguments\_list\_1(self, p):

"""actual\_arguments\_list : actual\_arguments\_list COMMA exp"""

print("""actual\_arguments\_list -> actual\_arguments\_list COMMA exp""")

def p\_actual\_arguments\_list\_2(self, p):

"""actual\_arguments\_list : exp"""

print("""actual\_arguments\_list -> exp""")

precedence = (

('nonassoc', 'LVALI'),

('nonassoc', 'LVAL'),

('nonassoc', 'BIOP'),

('nonassoc', 'COMOP'),

('nonassoc', 'BITOP'),

('left', 'IF'),

('left', 'ELSEIF'),

('left', 'ELSE'),

('left', 'COMMA'),

('left', 'ASSIGNMENT'),

('left', 'OR'),

('left', 'AND'),

('left', 'NOT'),

('left', 'BITWISE\_OR'),

('left', 'BITWISE\_AND'),

('left', 'BITWISE\_NOT'),

('left', 'LE', 'EQ', 'NE', 'GE', 'GT', 'LT'),

('left', 'SHIFT\_LEFT', 'SHIFT\_RIGHT'),

('left', 'ADDITION', 'SUBTRACTION'),

('left', 'MULTIPLICATION', 'DIVISION'),

('left', 'POWER'),

('left', 'MODULO'),

('left', 'UMINUS'),

('left', 'RP', 'LP')

)

def build(self, \*\*kwargs):

"""build the parser"""

self.parser = yacc.yacc(module=self, \*\*kwargs)

return self.parser