دانشکده مهندسی کامپیوتر

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

فاز اول پروژه ی درس کامپایلر

استاد درس: دکتر رزازی

زهرا نژادیان ۹۴۳۳۱۰۵- سپیده ملانوروزی ۹۴۳۱۰۷۲

تاریخ تحویل : دوشنبه ۲۰ آبان – موعد تحویل : دوشنبه ۲۰ 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 :

```
THIS IS SYMBOL TABLE
Program  0
var      1
integerId 2
_main    3
boolean  4
var_2    5
_func    6
i        7
j        8
```

```
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
```

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

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

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
```