Федеральное государственное автономное
образовательное учреждение
высшего профессионального образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт Космических и Информационных Технологий

Кафедра Информатики

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №4

**Реализация блока генерации промежуточного
кода транслятора простого языка программирования**

Преподаватель __________ А.С. Кузнецов
подпись, дата

Студент гр. КИ10-11 031010132 _____________ К.О. Васильев
подпись, дата

Красноярск 2014

1 Цель работы

Изучение методов генерации промежуточного кода с их программной реализацией.

2 Постановка задачи

Входной язык содержит операторы условий *if* с необязательной частью *else*, операторы цикла с параметром, операторы циклов с пред- и постусловием, арифметические выражения, разделенные символом точки с запятой (';'), операторы объявления переменных целого (*binary*, *hexadecimal*) и вещественного типов и их массивов. В операторах могут использоваться идентификаторы, целочисленные константы в шестнадцатеричной системе, битовые строки (начинающиеся с обязательной пары знаков 0*b* или 0*B* последовательности из 0 и 1), десятичные числа с плавающей точкой, знаки присваивания(':=' и другие), инкремента (‘++’), декремента (‘--’), элементы массивов, знаки операций и круглые скобки.

Операции: сравнения (<, >, =, ==, <>, !=, >=, <=), численная, логическая и побитовая арифметика; обращение к элементам массивов по индексу.

3 Исходный текст распознавателя

3.1 Спецификация лексического анализатора

```
# type names
SEMICOLON = 'SEMICOLON'
COMMA = 'COMMA'
COLON = 'COLON'
ID = 'ID'
FOR = 'FOR'
IF = 'IF'
ELSE = 'ELSE'
WHILE = 'WHILE'
DO = 'DO'
BINARY = 'BINARY'
HEXADECIMAL = 'HEXADECIMAL'
FLOAT = 'FLOAT'
BREAK = 'BREAK'
CONTINUE = 'CONTINUE'
PRINT = 'PRINT'
READF = 'READF'
READI = 'READI'
HCONST = 'HCONST'
BCONST = 'BCONST'
FCONST = 'FCONST'
ASSIGN = 'ASSIGN'
GREATER = 'GREATER'
LESS = 'LESS'
```

```
EQUALS = 'EQUALS'
NOTEQUALS = 'NOTEQUALS'
GREQUALS = 'GREQUALS'
LSEQUALS = 'LSEQUALS'
PLUS = 'PLUS'
MINUS = 'MINUS'
MULTIPLY = 'MULTIPLY'
DIVIDE = 'DIVIDE'
INR = 'INR'
DCR = 'DCR'
LSUM = 'LSUM'
LMUL = 'LMUL'
XOR = 'XOR'
LPAR = 'LPAR'
RPAR = 'RPAR'
LBRACE = 'LBRACE'
RBRACE = 'RBRACE'
LBRACKET = 'LBRACKET'
RBRACKET = 'RBRACKET'


builtins = (PRINT, READF, READI)

keywords = (FOR, BINARY, HEXADECIMAL, FLOAT, IF, ELSE, WHILE, DO,
BREAK, CONTINUE)

# reserved words
reserved = builtins + keywords

punctuators = (
    SEMICOLON,
    COMMA,
    COLON,
    LPAR,
    RPAR,
    LBRACE,
    RBRACE,
    LBRACKET,
    RBRACKET
)

operators = (
    ASSIGN,
    GREATER,
    LESS,
    EQUALS,
    NOTEQUALS,
    GREQUALS,
    LSEQUALS,
    INR,
    DCR,
    PLUS,
    MINUS,
    MULTIPLY,
    DIVIDE,
    LSUM,
    LMUL,
    XOR
)

ids = (
    ID,
)

constants = (
    HCONST,
    BCONST,
    FCONST
)

tokens = reserved + punctuators + operators + ids + constants

t_ignore = ' \t'


def t_NEWLINE(t):
    r'\n+'
    t.lexer.lineno += t.value.count('\n')


t_COMMA = r','
t_SEMICOLON = r';'
t_COLON = r':'
t_ASSIGN = r':='
t_LPAR = r'\('
t_RPAR = r'\)'
t_LBRACE = r'{'
t_RBRACE = r'}'
t_LBRACKET = r'\['
t_RBRACKET = r'\]'
t_HCONST = r'0x[0-9a-fA-F]+'
t_BCONST = r'0(B|b)[01]+'
t_FCONST = r'[0-9]*\.[0-9]+'

#arithmetic operations
t_MINUS = r'-'
t_PLUS = r'\+'
t_MULTIPLY = r'\*'
t_DIVIDE = r'/'
t_INR = r'\+\+'
t_DCR = r'--'
t_LSUM = r'\|'
```

```
t_LMUL = r'&'
t_XOR = r'\^'

# comparison operators
t_GREATER = r'>'
t_LESS = r'<'
t_EQUALS = r'==?'
t_NOTEQUALS = r'(!=|<>)'
t_GREQUALS = r'>='
t_LSEQUALS = r'<='

reserved_map = {}
for r in reserved:
    reserved_map[r.lower()] = r


def t_ID(t):
    r'[A-Za-z_][\w_]*'
    t.type = reserved_map.get(t.value, ID)
    return t

from ply.lex import LexError
class IllegalTokenException(LexError):
    def __init__(self, character, line_number):
        self.character = character
        self.line_number = line_number

    def __str__(self):
        return "Illegal character '{0}' at line
{1}".format(self.character, self.line_number)


def t_error(t):
    t.lexer.skip(1)
    print("Illegal character '{0}' at line {1}".format(t.value[0],
t.lineno))
    #raise IllegalTokenException(t.value[0], t.lineno)


def type_str(t):
    if t in reserved:
        return 'KEYWORD'
    if t in punctuators:
        return 'PUNCTUATOR'
    if t in operators:
        return 'OPERATOR'
    if t in ids:
        return 'ID'
    if t in constants:
        return 'CONSTANT'
```

```
def token_str(token):
    type = (str(type_str(token.type)) + ':').ljust(12, ' ')
    val = str(token.value).ljust(12, ' ')
    line = token.lineno
    return '{0}\t{1}\tline:{2}'.format(type, val, line)

import ply.lex as lex
lxr = lex.lex()
#a = dict(lxr)

def test(code, token_callback=None, error_callback=None):
    lxr.lineno = 1
    lxr.input(code)
    tokens = []
    errors = []
    while True:
        try:
            tok = lxr.token()
            if not tok:
                break
        except LexError as e:
            if error_callback:
                error_callback(e)
            errors.append(e)
        else:
            if token_callback:
                token_callback(tok)
            tokens.append(tok)
    return tokens, errors
```

3.2 Спецификация блока семантического анализа и построения 3-адресного кода

```
import ply.yacc as yacc
import copy
import itertools

from lexer import tokens, lxr
from node import Node, Leaf
from variable_type import VariableType, FunctionType
from bytecode_formatter import FormatterState, ConditionLabels

precedence = (
    ('nonassoc', 'LESS', 'GREATER', 'GREQUALS', 'LSEQUALS'),
    ('right', 'ASSIGN'),
    ('right', 'PLUS', 'MINUS'),
    ('right', 'MULTIPLY', 'DIVIDE'),
    ('right', 'INR', 'DCR'),
    ('left', 'NOTEQUALS', 'EQUALS'),
    ('right', 'UMINUS')
)

defined_vars = {}
errors = []

bytecode_state = FormatterState()


def p_program_start(p):
    'program : compound_statement'
    p[0] = Node(p, [p[1]])


def p_statement_list(p):
    '''statement_list : statement SEMICOLON statement_list
                        | empty'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = Node(p, [p[1], Leaf(p, 2), p[3]])
        bytecode_state.reset_state()


def p_nonterminated_statement(p):
    '''statement_list : statement error statement_list'''
                     errors.append('Missing     semicolon,     line
{0}'.format(p.lineno(2)))
    p[0] = Node(p, [p[1], p[3]])


def p_statement(p):
    '''statement : expression
                 | for_statement
                 | compound_statement
                 | if_statement
                 | while_statement
                 | dowhile_statement
                 | assign_statements'''
    p[0] = Node(p, [p[1]])
    p[0].type = p[1].type


def p_compound_statement(p):
    'compound_statement : LBRACE statement_list RBRACE'
    p[0] = Node(p, [Leaf(p, 1), p[2], Leaf(p, 3)])


def p_expression_numeric(p):
    'expression : numeric_expression'
    p[0] = p[1]
```

```
def p_expression_constants(p):
    '''numeric_expression : id_expression
                          | letters'''
    p[0] = p[1]


def p_id_expression(p):
    'id_expression : ID'
    if p.slice[1].value not in defined_vars:
          errors.append('Attempt to use undefined variable \'{0}\'
at line {1}'.format(p[1], p.lineno(1)))
                    p[0] = Node(p, leaf=Leaf(p, 1),
type=VariableType.get_type(p[1], defined_vars))


def p_literals_expression(p):
    '''letters : HCONST
               | BCONST
               | FCONST'''
                    p[0] = Node(p, leaf=Leaf(p, 1),
type=VariableType.get_type(p.slice[1].type, defined_vars))
    if p.slice[1].type == 'BCONST':
        p[0].type.var_name = str(int(p[1], 2))
    elif p.slice[1].type == 'HCONST':
        p[0].type.var_name = str(int(p[1], 16))
    else:
        p[0].type.var_name = p[1]


def p_expression_comparison_operations(p):
    '''expression : expression LESS expression
                  | expression GREATER expression
                  | expression GREQUALS expression
                  | expression LSEQUALS expression
                  | expression NOTEQUALS expression
                  | expression EQUALS expression'''
         if not VariableType.can_cast(VariableType.get_type(p[1],
defined_vars), VariableType.get_type(p[3], defined_vars)):
              errors.append('Cannot perform \'{0}\' operation on
different types at line {1}'.format(p[2], p.lineno(2)))
    p[0] = Node(p, [p[1], Leaf(p, 2), p[3]],
                          type=VariableType(VariableType.type_bool,
var_name=bytecode_state.reserve_var()))
               bytecode_state.code += '{0} := {1} {2}
{3}\n'.format(p[0].type.var_name, p[1].type.var_name, p[2],
                                                          p[3].type
.var_name)


def p_expression_arithmetic_operations(p):
    '''numeric_expression : expression PLUS expression
                  | expression MINUS expression
                  | expression MULTIPLY expression
                  | expression DIVIDE expression
                  | MINUS expression %prec UMINUS'''
    expr_temp_var = bytecode_state.reserve_var()
    if len(p) > 3:
         if not (VariableType.can_cast(VariableType.get_type(p[1],
defined_vars), VariableType.type_binary)
                                                                  and
VariableType.can_cast(VariableType.get_type(p[3], defined_vars),
VariableType.type_binary)):
            errors.append(
                 'Cannot perform \'{0}\' operation for non-numeric
types at line {1}'.format(p[2], p.lineno(2)))
                  p[0] = Node(p, [p[1], Leaf(p, 2), p[3]],
type=VariableType.get_type(p[1], defined_vars))
                    bytecode_state.code += '{0} := {1} {2}
{3}\n'.format(expr_temp_var, p[1].type.var_name, p[2],
                                                                p[3].
type.var_name)
    else:
         if not (VariableType.can_cast(VariableType.get_type(p[2],
defined_vars), VariableType.type_binary)):
                errors.append('Cannot perform unary minus at line
{0}'.format(p.lineno(1)))
                       p[0] = Node(p, [Leaf(p, 1), p[2]],
type=VariableType.get_type(p[1], defined_vars))
                        bytecode_state.code += '{0} := {1}
{2}\n'.format(expr_temp_var, p[1], p[2].type.var_name)
    p[0].type.var_name = expr_temp_var


def p_expression_bool_arithmetic(p):
    '''numeric_expression : expression LSUM expression
                  | expression LMUL expression
                  | expression XOR expression'''
      if VariableType.can_cast(p[1].type, VariableType.type_float,
False) or \
                                   VariableType.can_cast(p[3].type,
VariableType.type_float, False):
           errors.append('Cannot perform logic operations on float
values, line {0}'.format(p.lineno(2)))
    p[0] = Node(p, [p[1], Leaf(p, 2), p[3]],
                type=p[1].type if VariableType.can_cast(p[1].type,
VariableType.type_hex) else p[3].type)
    expr_temp_var = bytecode_state.reserve_var()
    p[0].type.var_name = expr_temp_var
               bytecode_state.code += '{0} := {1} {2}
{3}\n'.format(expr_temp_var, p[1].type.var_name, p[2],
                                                            p[3].type
.var_name)


def p_exression_uoperation(p):
    '''numeric_expression : INR expression
                  | DCR expression'''
    if not p[2].leaf or p[2].leaf.type != 'ID':
          errors.append('Cannot perform \'{0}\' operation at line
{1}'.format(p[1], p.lineno(1)))
    p[0] = Node(p, [Leaf(p, 1), p[2]], type=p[2].type)
               bytecode_state.code += '{0} := {0} {1}
1\n'.format(p[2].type.var_name, p[1][0])


def p_different_assign_statement(p):
    '''assign_statements : assign_statement
                         | declare'''
    p[0] = p[1]


def p_statement_assign(p):
        'assign_statement : id_expression array_indexes ASSIGN
expression'
    id_type = copy.deepcopy(p[1].type)
    if len(p[2].children) > len(id_type.array_dimensions):
        errors.append('Attempt to access too deep into array, line
{0}'.format(p.lineno(3)))
    elif p[2].children:
                                 id_type.array_dimensions =
id_type.array_dimensions[len(p[2].children):]
            if not id_type.can_cast(VariableType.get_type(p[4],
defined_vars)):
              errors.append('Cannot perform \'{0}\' operation on
different types at line {1}'.format(p[3], p.lineno(3)))
    p[0] = Node(p, [p[1], p[2], Leaf(p, 3), p[4]])
    if p[2].children:
                  array_index = calculate_index(p[2].children,
p[1].type.array_dimensions)

        if p[4].type.array_dimensions:
            def get_array_elements(el):
                if el.expr == 'array_declare':
                    elements = []
                    for c in el.children:
                        cur_el = get_array_elements(c)
                        if isinstance(cur_el, list):
                            elements.extend(cur_el)
                        else:
                            elements.append(cur_el)
                    return elements
                return el.type.var_name
            els = get_array_elements(p[4].children[0])
        else:
            els = [p[4].type.var_name]
        for id, element in enumerate(els):
                            bytecode_state.code  +=  '{0}[{1}]  :=
{2}\n'.format(p[1].type.var_name, array_index, element)
            if id < len(els) - 1:
                next_array_index = bytecode_state.reserve_var()
                           bytecode_state.code  +=  '{0}  :=  {1}  +
1\n'.format(next_array_index, array_index)
                array_index = next_array_index
    else:
                             bytecode_state.code   +=   '{0}   :=
{1}\n'.format(p[1].type.var_name, p[4].type.var_name)


def p_statement_declare(p):
    '''declare : type ID ASSIGN expression'''
    p[0] = Node(p, [p[1], Leaf(p, 2), Leaf(p, 3), p[4]])
    if p[2] not in defined_vars:
        cur_variable_type = copy.deepcopy(p[4].type)
        cur_variable_type.var_name = p[2]
        defined_vars[p[2]] = cur_variable_type
    t1 = p[1].type
    t2 = p[4].type
    t2_name = t2.var_name[:]
    if isinstance(t2.array_dimensions, list):
        t2_array_dimensions = t2.array_dimensions[:]
    else:
        t2_array_dimensions = t2.array_dimensions
    t2.array_dimensions = t1.array_dimensions
     if not t1.can_cast(t2, False) or (t1.array_dimensions is not
None and t2_array_dimensions is not None and
                                               t1.array_dimensions
!= len(t2_array_dimensions)):
             errors.append('Cannot perform assign operation with
different types at line {0}'.format(p.lineno(3)))
    bytecode_state.code += '{0} := {1}\n'.format(p[2], t2_name)


def p_type(p):
    '''type : basic_type multiple_stars'''
    p[0] = Node(p, [p[1]], type=p[1].type)
    p[0].type.array_dimensions = p[2]


def p_basic_type(p):
    '''basic_type : BINARY
                  | HEXADECIMAL
                  | FLOAT'''
    p[0] = Node(p, leaf=Leaf(p, 1), type=VariableType(p[1]))
```

```
def p_multiple_brackets(p):
    '''multiple_stars : MULTIPLY multiple_stars
                         | empty'''
    if len(p) == 2:
        p[0] = 0
    else:
        p[0] = p[2] + 1


def p_expression_array_declare(p):
    'expression : array_declare'
    p[0] = Node(p, [p[1]], type=copy.deepcopy(p[1].type))
    array_name = bytecode_state.reserve_var()
    p[0].type.var_name = array_name[:]

    def define_array(arr, var_id):
        if arr.expr == 'array_declare':
            for item in arr.children:
                var_id = define_array(item, var_id)
        else:
                            bytecode_state.code += '{0}[{1}] :=
{2}\n'.format(array_name, var_id, arr.type.var_name)
            return var_id + 1
        return var_id

    define_array(p[1], 0)


def p_array_declare(p):
      '''array_declare : LBRACKET array_element array_params_list
RBRACKET
                     | LBRACKET RBRACKET'''
    p[0] = Node(p, [], type=VariableType(None, [0]))
    if len(p) > 3:
        p[0].children.append(p[2])
        if p[3].children:
            p[0].children.extend(p[3].children)

        first_type = p[0].children[0].type
        for param in p[0].children:
            if not first_type.can_cast(param.type, True):
                       errors.append('Nested array variables have
different types, line {0}'.format(p.lineno(1)))
                break
        p[0].type = first_type
        if p[0].type.array_dimensions is None:
            p[0].type.array_dimensions = []
        p[0].type.array_dimensions.insert(0, len(p[0].children))
```

```
def p_array_inner(p):
    '''array_element : numeric_expression
                     | array_declare'''
    p[0] = p[1]


def p_array_params(p):
    '''array_params_list : COMMA array_element array_params_list
                         | empty'''
    if len(p) == 4:
        p[0] = Node(p, [p[2]])
        if p[3].children:
            p[0].children.extend(p[3].children)
    else:
        p[0] = p[1]


def p_def_array_access(p):
    'expression : array_access'
    p[0] = p[1]


def p_array_element_access(p):
    '''array_access : expression index array_indexes'''
    access_depth = 1 + len(p[3].children)
    vartype = copy.deepcopy(p[1].type)
    if access_depth > len(vartype.array_dimensions):
        errors.append('Attempt to access too deep into array, line
{0}'.format(p.lineno(2)))
         vartype.array_dimensions   =   vartype.array_dimensions[:-
access_depth]
    vartype.var_name = bytecode_state.reserve_var()
    p[0] = Node(p, [p[1], p[2]], type=vartype)
    if p[3].children:
        p[0].children.extend(p[3].children)

             access_index   =   calculate_index(p[0].children[1:],
p[1].type.array_dimensions)
                  bytecode_state.code    +=    '{0}    :=    {1}
[{2}]\n'.format(vartype.var_name, p[1].type.var_name,
                                                      access_index)


def calculate_index(index_arrays, dimensions):
    if len(index_arrays) == 1:
        return index_arrays[-1].type.var_name

    first_item = True
    cur_idx = 0
    for idx in index_arrays[:-1]:
```

```
                      bytecode_state.code += '{0} := {1} *
{2}\n'.format(bytecode_state.reserve_var(), idx.type.var_name,
                                                            dimensi
ons[cur_idx])
        cur_idx += 1
        if not first_item:
            var_nums = bytecode_state.temp_var_number
                         bytecode_state.code += '{0} := {1} +
{2}\n'.format(bytecode_state.reserve_var(),
                                                                byt
ecode_state.temp_var(var_nums - 1),
                                                                byt
ecode_state.temp_var(var_nums - 2))
        else:
            first_item = False
                bytecode_state.code += '{0} := {1} +
{2}\n'.format(bytecode_state.reserve_var(),
                                                        bytecode_st
ate.temp_var(bytecode_state.temp_var_number - 2),
                                                        index_array
s[-1].type.var_name)
    return bytecode_state.last_var()


def p_array_access_list(p):
    '''array_indexes : index array_indexes
                     | empty'''
    if len(p) == 2:
        #empty
        p[0] = p[1]
    else:
        p[0] = Node(p, [p[1]])
        if p[2].children:
            p[0].children.extend(p[2].children)


def p_array_element(p):
    'index : LBRACKET numeric_expression RBRACKET'
    if not p[2].type.can_cast(VariableType.type_binary, False):
          errors.append('Array indexes should be integer numbers,
line {0}'.format(p.lineno(1)))
    p[0] = p[2]


def p_expression_group(p):
    'expression : LPAR expression RPAR'
    p[0] = Node(p, [Leaf(p, 1), p[2], Leaf(p, 3)], type=p[2].type)


def p_missing_rpar_error(p):
    '''expression : LPAR expression error'''
```

```
            errors.append('Missing closing parenthesis, line
{0}'.format(p.lineno(3)))
    p[0] = Node(p, [Leaf(p, 1), p[2]])


def p_loop_enter(p):
    'loop_enter :'
    bytecode_state.enter_loop()
                                     bytecode_state.code +=
bytecode_state.current_loop().label_start + ':\n'


def p_loop_leave(p):
    'loop_leave :'
     bytecode_state.code += bytecode_state.current_loop().label_end
+ ':\n'
    bytecode_state.leave_loop()


def p_for_statement(p):
      'for_statement : FOR LPAR for_declare SEMICOLON loop_enter
for_cond SEMICOLON for_next RPAR compound_statement loop_leave'
    p[0] = Node(p,
                  [Leaf(p, 1), Leaf(p, 2), p[3], Leaf(p, 4), p[6],
Leaf(p, 7), p[8],
                 Leaf(p, 9), p[10]])


def p_for_declare_part(p):
    '''for_declare : assign_statements
                   | empty'''
    p[0] = p[1]


def p_for_cond(p):
    '''for_cond : expression
                | empty'''
    if p.slice[1].type == 'expression':
                        if not VariableType.can_cast(p[1].type,
VariableType.type_bool):
                 errors.append('Conditional part of for statement
should be of bool type, line {0}'.format(p.lineno(1)))
    p[0] = Node(p, [p[1]], type=p[1].type)
               bytecode_state.code += 'iffalse {0} goto
{1}\n'.format(bytecode_state.last_var(),
                                                            bytecod
e_state.current_loop().label_end)


def p_for_expr(p):
    '''for_next : expression
```

```
                | empty'''
    p[0] = p[1]


def p_while_statement(p):
      'while_statement : WHILE loop_enter while_condition COLON
compound_statement loop_leave'
         if not VariableType.can_cast(VariableType.get_type(p[3],
defined_vars), VariableType.type_bool):
                 errors.append('Expression mus be boolean, line
{0}'.format(p.lineno(1)))
    p[0] = Node(p, [Leaf(p, 1), p[3], Leaf(p, 4), p[5]])


def p_while_condition(p):
    'while_condition : expression'
    p[0] = Node(p, [p[1]], type=p[1].type)
               bytecode_state.code += 'iffalse {0} goto
{1}\n'.format(bytecode_state.last_var(),
                                                                      bytecod
e_state.current_loop().label_end)


def p_dowhile_statement(p):
      'dowhile_statement : DO loop_enter compound_statement WHILE
dowhile_condition loop_leave'
         if not VariableType.can_cast(VariableType.get_type(p[5],
defined_vars), VariableType.type_bool):
                 errors.append('Expression mus be boolean, line
{0}'.format(p.lineno(3)))
    p[0] = Node(p, [Leaf(p, 1), p[3], Leaf(p, 4), p[5]])


def p_dowhile_condition(p):
    'dowhile_condition : expression'
    p[0] = Node(p, [p[1]], type=p[1].type)
                bytecode_state.code += 'iftrue {0} goto
{1}\n'.format(bytecode_state.last_var(),
                                                                     bytecode
_state.current_loop().label_start)


def p_break_continue_statement(p):
    '''statement : BREAK
                 | CONTINUE'''
    p[0] = Node(p, leaf=Leaf(p, 1))
    if not bytecode_state.current_loop():
              errors.append('{0} should be nested in loop, line
{1}'.format(p[1].title(), p.lineno(1)))
    else:
        bytecode_state.code += 'goto {0}\n'.format(
                        bytecode_state.current_loop().label_end if
p.slice[1].type == 'BREAK'
            else bytecode_state.current_loop().label_start)


def p_if_statement(p):
      '''if_statement : IF expression enter_if COLON true_branch
false_branch
                    | IF expression enter_if COLON true_branch'''
         if not VariableType.can_cast(VariableType.get_type(p[2],
defined_vars), VariableType.type_bool):
                errors.append('Expression must be boolean, line
{0}'.format(p.lineno(1)))
    p[0] = Node(p, [Leaf(p, 1), p[2], Leaf(p, 4), p[5]])
    if len(p) == 7:
        p[0].children.append(p[6])
                                     bytecode_state.code +=
bytecode_state.condition_stack.pop().label_end + ':\n'


def p_enter_if(p):
    'enter_if :'
     cond_labels = ConditionLabels(bytecode_state.reserve_label(),
bytecode_state.reserve_label())
    bytecode_state.condition_stack.append(cond_labels)
               bytecode_state.code += 'iffalse {0} goto
{1}\n'.format(bytecode_state.last_var(),
                                                            cond_la
bels.label_false)


def p_true_branch(p):
    'true_branch : compound_statement'
    p[0] = Node(p, [p[1]])
    if bytecode_state.condition_stack[-1].label_false:
                                 bytecode_state.code += 'goto
{0}\n'.format(bytecode_state.condition_stack[-1].label_end)
           bytecode_state.code += bytecode_state.condition_stack[-
1].label_false + ':\n'


def p_false_branch(p):
    '''false_branch : ELSE COLON compound_statement'''
    p[0] = Node(p, [Leaf(p, 1), Leaf(p, 2), p[3]])


def p_func_call(p):
    '''expression : func LPAR func_params RPAR'''
    if len(p[1].type.params) < len(p[3].children):
           errors.append('Attempt to call function with too much
arguments, line {0}'.format(p.lineno(2)))
```

```
    elif len(p[1].type.params) > len(p[3].children):
          errors.append('Attempt to call function with not enough
arguments, line {0}'.format(p.lineno(2)))
            elif   any(itertools.starmap(lambda   el1,   el2:   not
el1.can_cast(el2),
                                zip(p[1].type.params, [c.type for c
in p[3].children]))):
          errors.append('Parameter types of function call does not
match declaration, line {0}'.format(p.lineno(2)))
                      p[0]     =     Node(p,     [p[1],     p[3]],
type=copy.deepcopy(p[1].type.return_type))
    p[0].type.var_name = bytecode_state.reserve_var()
              bytecode_state.code   +=   '{0}   :=   call   {1},
{2}\n'.format(p[0].type.var_name,                  p[1].leaf.value,
len(p[3].children))


def p_func_params(p):
    '''func_params : expression func_params_list
                   | empty'''
    p[0] = Node(p, [])
    if len(p) > 2:
        p[0].children.append(p[1])
        if p[2].children:
            p[0].children.extend(p[2].children)
        for param in p[0].children:
                                  bytecode_state.code   +=   'param
{0}\n'.format(param.type.var_name)


def p_func_params_list(p):
    '''func_params_list : COMMA expression func_params_list
                        | empty'''
    if len(p) > 2:
        p[0] = Node(p, [p[2]])
        if p[3].children:
            p[0].children.extend(p[3].children)
    else:
        p[0] = p[1]


def p_func_print(p):
    """func : PRINT"""
                    p[0]     =     Node(p,     leaf=Leaf(p,     1),
type=FunctionType(VariableType(None),
[VariableType(VariableType.type_any)]))


def p_func_readi(p):
    """func : READI"""
                    p[0]     =     Node(p,     leaf=Leaf(p,     1),
type=FunctionType(VariableType(VariableType.type_hex)))


def p_func_readf(p):
    """func : READF"""
        p[0] = Node(p, leaf=Leaf(p, 1),
type=FunctionType(VariableType(VariableType.type_float)))


def p_empty(p):
    'empty :'
    p[0] = Node(p)


def p_error(p):
    if not p:
        print "Syntax error at EOF"
    else:
        errors.append('Unexpected symbol \'{0}\' at line
{1}'.format(p.value, p.lineno))
        yacc.errok()
        return yacc.token()


yacc_parser = yacc.yacc(debug=True)


def yparse(data, debug=0):
    yacc_parser.error = 0
    lxr.lineno = 1
    p = yacc_parser.parse(data, debug=debug, lexer=lxr)
    if yacc_parser.error:
        return None
    return p, bytecode_state.code, errors
```

3.3 Исходный код файла node.py

```
class Leaf(object):
    def __init__(self, parse, id):
        self.type = parse.slice[id].type
        self.value = parse.slice[id].value

    def __str__(self):
        return "{0} [{1}:'{2}']".format(self.type, 'name' if
self.type in ids + reserved else 'text', self.value)

    def to_ast(self):
        return str(self) + '\n'

    def ast(self):
        return str(self)
```

```
class Node(object):
    def __init__(self, prod, children=None, leaf=None, type=None):
        self.expr = prod.slice[0].type
        if children:
            self.children = children
        else:
            self.children = []
        self.leaf = leaf
        self.type = type

    def _get_statements(self):
        if self.expr != 'statement_list' or len(self.children) < 3:
            return None
        statements = [self.children[0]]
        child_st = self.children[2]._get_statements()
        if child_st:
            statements.extend(child_st)
        return statements

    def to_tree(self):
          cur = '{0}/{1}\n'.format(self.expr, 1 if self.leaf else
len(self.children))
        child_num = 1
        child_text = ''
        if self.leaf:
            child_text = '{0} {1}'.format(child_num, self.leaf)
        else:
            for c in self.children:
                child_text += '{0} {1}\n'.format(child_num, c)
                child_num += 1
         cur += '\n'.join('  ' + s for s in child_text.split('\n')
if s != '') + '\n'
        return cur

    def _append_to(self, txt, val='  '):
        append_to_end = ''
        if txt.endswith('\n'):
            append_to_end = '\n'
         res = '\n'.join(val + s for s in txt.split('\n') if s !=
'') + append_to_end
        return res

    def ast(self):
        if self.expr == 'program':
            return self.children[0].ast()
        if self.expr == 'compound_statement':
            self.expr = 'S'
            self.children = [c.ast() for c in self.children[1:-1]]
            return self
        if self.expr == 'statement_list':
            self.children = self._get_statements()
            self.expr = 'L'
            self.children = [c.ast() for c in self.children]
            return self
        if self.expr == 'expression':
                       if hasattr(self.children[0], 'value') and
self.children[0].type == 'LPAR':
                return self.children[1].ast()
            if len(self.children) == 3:
                self.expr = self.children[1].value
                del self.children[1]
                self.children = [c.ast() for c in self.children]
                return self
            if len(self.children) == 2:
                self.expr = 'call'
                self.children.extend(self.children[1].children)
                del self.children[1]
                self.children = [c.ast() for c in self.children]
                return self
        if self.expr == 'numeric_expression':
            if len(self.children) == 2:
                self.expr = self.children[0]
                              self.children = [c.ast() for c in
self.children[1:]]
                return self
            if len(self.children) == 3:
                self.expr = self.children[1]
                del self.children[1]
                self.children = [c.ast() for c in self.children]
                return self
        if self.expr == 'array_declare':
            self.expr = 'array'
            self.children = [c.ast() for c in self.children]
        if self.expr == 'array_access':
            self.expr = 'index'
            self.children = [c.ast() for c in self.children]
            return self
        if self.expr == 'index':
            return self.children[1].ast()
        if self.expr == 'assign_statement':
            self.expr = self.children[2]
            del self.children[2]
            self.children = [c.ast() for c in self.children]
            return self
        if self.expr == 'declare':
            self.expr = self.children[2]
            del self.children[2]
            self.children = [c.ast() for c in self.children[1:]]
            return self
        if self.expr == 'for_statement':
            self.expr = 'C'
             self.children = [c.ast() for c in [self.children[idx]
for idx in (2, 4, 6, 8)]]
            return self
             if self.expr == 'while_statement' or self.expr ==
'dowhile_statement':
            self.expr = 'C'
             self.children = [c.ast() for c in [self.children[idx]
for idx in (1, 3)]]
            return self
                if self.expr in ('for_cond', 'while_condition',
'dowhile_condition'):
            self.expr = 'cond'
            self.children = [c.ast() for c in self.children]
            return self
        if self.expr == 'if_statement':
            self.expr = 'flow'
            del self.children[0]
            del self.children[1]
            self.children = [c.ast() for c in self.children]
            return self
        if self.expr == 'true_branch':
            self.children = [c.ast() for c in self.children]
            return self
        if self.expr == 'false_branch':
            self.children = [self.children[-1].ast()]
            return self
        if len(self.children) == 1:
            return self.children[0].ast()
        if self.leaf:
            return self.leaf.ast()
        return self


    def __str__(self):
        return self.to_tree()
```

4 Тестовые примеры работы программы

Для кода на ЯВУ:

```
{
	for (binary a := 0b0; a < 0xf; ++a)
	{
		print(a);
	};
}
```

3-адресный код:

```
a := 0
@L1:
@t1 := a < 15
iffalse @t1 goto @L2
a := a + 1
param a
call print, 1
@L2:
```

Для следующего кода:

```
{
	hexadecimal count := 0x0;
	while readi() < 0x99:
	{
		++count;
	};
}
```

Соответствующий 3-адресный код:

```
count := 0
@L1:
@t1 := call readi, 0
@t2 := @t1 < 153
iffalse @t2 goto @L2
count := count + 1
@L2:
```

Для кода на ЯВУ:

```
{
	print(readf() >= readf());
}
```

3-адресный код:

```
@t1 := call readf, 0
@t2 := call readf, 0
```

```
@t3 := @t1 >= @t2
param @t3
call print, 1
```

Код на ЯВУ:

```
{
	hexadecimal numb := readi();
	binary truncator := 0xf;
	if numb & truncator > 0xd:
	{
		numb := --numb + truncator;
	}
	else:
	{
		numb := truncator;
	};
}
```

3-адресный код:

```
@t1 := call readi, 0
numb := @t1
truncator := 15
@t2 := numb & truncator
@t3 := @t2 > 13
iffalse @t3 goto @L2
numb := numb - 1
@t4 := numb + truncator
numb := @t4
goto @L1
@L2:
numb := truncator
@L1:
```