PROGRAMMING LANGUAGE TRANSLATION

PHASE 3

JAVA BYTE CODE GENERATION

SANDRA SHERIF WASFI 3940

RIMON ADEL KARAM 4441

YOUSSEF MOHAMED 3812

ADEL ATEF MELEKA 3960

OBJECTIVE

This phase of the assignment aims to practice techniques of constructing semantics rules to
generate intermediate code.

PROBLEM DESCRIPTION

Generated bytecode must follow Standard bytecode instructions defined in Java Virtual Machine Specification
http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html

http://en.wikipedia.org/wiki/Java_bytecode

Proposed grammars are required to cover the following features:

- Primitive types (**int**, **float**) with operations on them (+, - , *, /)
- Boolean Expressions (Bonus marks)
- Arithmetic Expressions
- Assignment statements
- **If-else** statements
- **for**loops (Bonus marks)
- **while** loops

FUNCTIONS OF ALL PHASES

PHASE 1

This phase represents the building of a lexical analyzer in a typical compiler. It takes regular expressions for different tokens is a specific language and produces a minimized NFA then this NFA is used in lexical analyzer to match different tokens.

PHASE 2

This phase represents the building of a syntax analyzer in a typical compiler. It takes grammatical rules and produces LL(1) parser for this grammar. LL(1) parses the tokens from lexical analyzer and produces parse tree.

PHASE 3

This phase represents the semantic analyzer in a typical compiler. It takes the parse tree then produces the intermediate code.

ALGORITHMS USED

Conditional expressions: in Boolean expressions we first generate two labels for true and false then we send them to the expression. After that the two labels are printed ahead of the two statements responsible of true and false.

Boolean Expressions: in Boolean expressions we generate the appropriate branch code using the inherited true and false attributes.

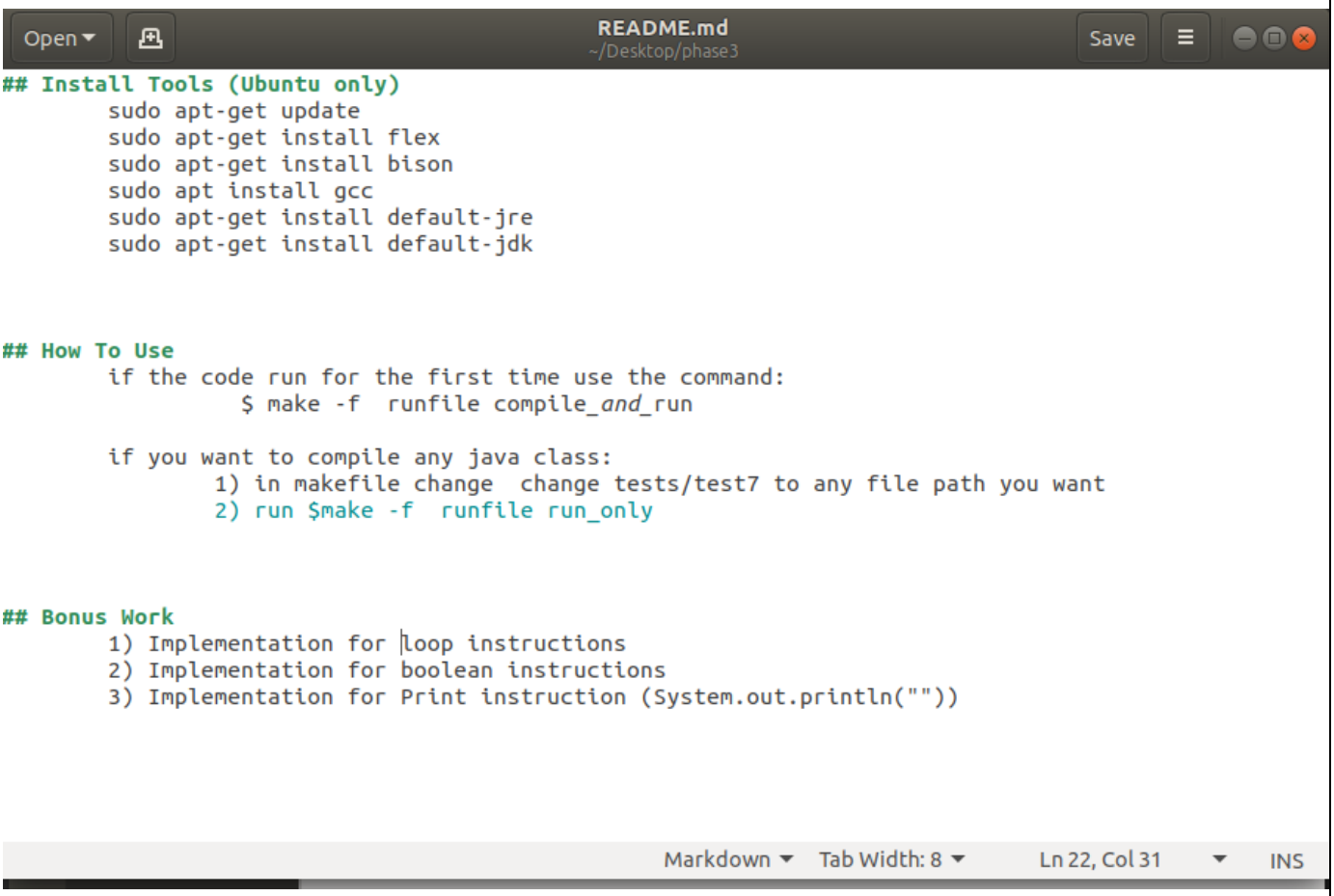
DATA STRUCTURES USED

In Bison, union directive specifies a union for every possible type of terminals and non-terminals.

For example, for constants of type **`int`** we put their type in the union as **int ival**.

Then we define a terminal as: **“%token INT_CONST”**, then we have terminal **INT_CONST** that has a semantic value of **int**.

INSTRUCTIONS AND HOW TO RUN



COMMENTS ABOUT USED TOOLS

In order to handle the required task, the main files are divided as follow:

1. **Lexical Analyzer (scanner.l):** in this phase we tokenize the input and define the meaning of every set of characters in the input code. It is done by ‘lex’ Tool.

2. **Syntax Analyzer (syntax.y) :** Rules are written in this file, then semantic actions is written in every rule. Here we write actions to generate Java Bytecode.
   This is done by ‘yacc [biscon]’ Tool.

3. **C++ code:** both files eventually generate C++ code that can be compiled with g++, so in both files you can put C++ code to enhance code generation.

ANY ASSUMPTIONS MADE

```
- assignments of the same types only
- no cond( true && expr ) | (true || expr)

//todo
strings print
++ operator
assignment in declaration
```

HELPER FUNCTIONS

Here is used Helper Functions:

```
// functions prototypes
void cast (string x, int type_t1);
void arith_cast(int from , int to, string op);
void relaCast(string op,char * nTrue, char * nFalse);
bool check_id(string id);
void write_class_header(void);  /* generate  header for class to be able to compile the code*/
void write_class_footer(void);  /* generate  footer for class to be able to compile the code*/
void add_code(string x);
void write_generated_code(void);
void define_var(string name, int type);
void back_patch(vector<int> *list, int num);
vector<int> * merge (vector<int> *list1, vector<int>* list2);
void printLineNumber(int num);
// end
```

SDT's that can be implemented during parsing can be characterized by introducing distinct marker nonterminal in place of each embedded action; each marker M has only one production, M -+ Eo If the grammar with marker nonterminal can be parsed by a given method, then the SDT can be implemented during parsing.

So, we used function **(marker)** that generate distinct labels to each semantic action.

SEMANTIC RULES

Here is the way the Sematic Rules are written in the Tool:

```
%token <ival> INT
%token <fval> FLOAT
%token <bval> BOOL
%token <idval> IDENTIFIER
%token <aopval> ARITH_OP
%token <aopval> RELA_OP
%token <aopval> BOOL_OP

%token IF_KEYWORD
%token ELSE_KEYWORD
%token WHILE_KEYWORD
%token FOR_KEYWORD

%token INT_KEYWORD
%token FLOAT_KEYWORD
%token BOOLEAN_KEYWORD

%token SEMI_COLON
%token EQUAL

%token OPEN_BRACKETS
%token CLOSE_BRACKETS
%token OPEND_PARENTHESIS
%token CLOSED_PARENTHESIS

%token SYSTEM_OUT

%type <sType> primitive_type
%type <expr_type> expression
%type <bexpr_type> b_expression
%type <stmt_type> statement
%type <stmt_type> statement_list
%type <stmt_type> if
%type <stmt_type> while
%type <stmt_type> for

%type <ival> marker
%type <ival> goto
```

```
%%
// jAVA CFG
method_body:
    {write_class_header();  }
    statement_list
    marker
    {
        back_patch($2.nextList,$3);
        write_class_footer();
    }
    ;
statement_list:
     statement
    |
    statement
    marker
    statement_list
    {
        back_patch($1.nextList,$2);
        $$.nextList = $3.nextList;
    }
    ;
marker:
{
    $$ = labelsCount;
    add_code(generate_label() + ":");
}
;
statement:
    declaration {vector<int> * v = new vector<int>(); $$.nextList =v;}
    |if {$$.nextList = $1.nextList;}
    |while  {$$.nextList = $1.nextList;}
    |for {$$.nextList = $1.nextList;}
    | assignment {vector<int> * v = new vector<int>(); $$.nextList =v;}
    | system_print {vector<int> * v = new vector<int>(); $$.nextList =v;}
    ;
```

```
declaration:
    primitive_type IDENTIFIER SEMI_COLON /* implement multi-variable declaration */
    {
        string str($2);
        if($1 == INT_T)
        {
            define_var(str,INT_T);
        }else if ($1 == FLOAT_T)
        {
            define_var(str,FLOAT_T);
        }
    }
    ;
primitive_type:
    INT_KEYWORD {$$ = INT_T;}
    | FLOAT_KEYWORD {$$ = FLOAT_T;}
    |BOOLEAN_KEYWORD {$$ = BOOL_T;}
    ;
goto:
{
    $$ = generated_code_list.size();
    add_code("goto ");
}
;
if:
    IF_KEYWORD OPEN_BRACKETS
    b_expression
    CLOSE_BRACKETS OPEND_PARENTHESIS
    marker
    statement_list
    goto
    CLOSED_PARENTHESIS
    ELSE_KEYWORD OPEND_PARENTHESIS
    marker
    statement_list
    CLOSED_PARENTHESIS
    {
        back_patch($3.trueList,$6);
        back_patch($3.falseList,$12);
        $$.nextList = merge($7.nextList, $13.nextList);
```

GRAMMAR RULES

The used Grammar rules are implemented in Sematic Rules. We used the same CFG as in Phase 2:

Java CFG

```
METHOD_BODY:: =   STATEMENT_LIST
STATEMENT_LIST::= STATEMENT |   STATEMENT_LIST   STATEMENT
STATMENT ::=      DECLARATION
                | IF
                | WHILE
                | ASSIGNMENT
DECLARATION ::= PRIMITIVE_TYPE IDENTIFIER;
PRIMITIVE_TYPE ::= int | float
IF ::= if ( EXPRESSION ) { STATEMENT} else { STATEMENT }
WHILE ::= while ( EXPRESSION ) { STATEMENT }
ASSIGNMENT ::= IDENTIFIER = EXPRESSION;
EXPRESSION ::= NUMBER
                | EXPRESSION INFIX_OPERATOR EXPRESSION
                | IDENTIFIER
                | ( EXPRESSION )
INFIX_OPERATOR ::= + | - | * | / | %| < | > | <= | >= | == | !=| |   | &&
```

BONUS WORK

1. Loop Instructions

```
while:
    marker
    WHILE_KEYWORD OPEN_BRACKETS
    b_expression
    CLOSE_BRACKETS OPEND_PARENTHESIS
    marker
    statement_list
    CLOSED_PARENTHESIS
    {
        add_code("goto " + get_Label($1));

        back_patch($8.nextList,$1);
        back_patch($4.trueList,$7);
        $$.nextList = $4.falseList;
    }
    ;
for:
    FOR_KEYWORD
    OPEN_BRACKETS
    assignment
    marker
    b_expression
    SEMI_COLON
    marker
    assignment
    goto
    CLOSE_BRACKETS
    OPEND_PARENTHESIS
    marker
    statement_list
    goto
    CLOSED_PARENTHESIS
    {
        back_patch($5.trueList,$12);
        vector<int> * v = new vector<int> ();
        v->push_back($9);
        back_patch(v,$4);
        v = new vector<int>();
        v->push_back($14);
        back_patch(v,$7);
        back_patch($13.nextList,$7);
        $$.nextList = $5.falseList;
    }
    ;
```

2. Boolean instructions

```
b_expression:
    BOOL
    {
        if($1)
        {   // bool is 'true'
            $$.trueList = new vector<int> ();
            $$.trueList->push_back(generated_code_list.size());
            $$.falseList = new vector<int>();
            add_code("goto ");
        }else
        {   //bool is 'flase'
            $$.trueList = new vector<int> ();
            $$.falseList= new vector<int>();
            $$.falseList->push_back(generated_code_list.size());
            add_code("goto ");
        }
    }
    |b_expression
    BOOL_OP
    marker
    b_expression
    {
        if(!strcmp($2, "&&"))
        {
            back_patch($1.trueList, $3);
            $$.trueList = $4.trueList;
            $$.falseList = merge($1.falseList,$4.falseList);
        }
        else if (!strcmp($2,"||"))
        {
            back_patch($1.falseList,$3);
            $$.trueList = merge($1.trueList, $4.trueList);
            $$.falseList = $4.falseList;
        }
    }
    | expression RELA_OP expression
    {
        string op ($2);
        $$.trueList = new vector<int>();
        $$.trueList ->push_back (generated_code_list.size());
        $$.falseList = new vector<int>();
        $$.falseList->push_back(generated_code_list.size()+1);
        add_code(get_operation(op)+ " ");
        add_code("goto ");
    }
```

3. Print Instructions

```
system_print:
    SYSTEM_OUT OPEN_BRACKETS expression CLOSE_BRACKETS SEMI_COLON
    {
        if($3.sType == INT_T)
        {
            /* expression is at top of stack now */
            /* save it at the predefined temp syso var */
            add_code("istore " + to_string(symbol_tab["1syso_int_var"].first));
            /* call syso */
            add_code("getstatic      java/lang/System/out Ljava/io/PrintStream;");
            /*insert param*/
            add_code("iload " + to_string(symbol_tab["1syso_int_var"].first ));
            /*invoke syso*/
            add_code("invokevirtual java/io/PrintStream/println(I)V");

        }else if ($3.sType == FLOAT_T)
        {
            add_code("fstore " + to_string(symbol_tab["1syso_float_var"].first));
            /* call syso */
            add_code("getstatic      java/lang/System/out Ljava/io/PrintStream;");
            /*insert param*/
            add_code("fload " + to_string(symbol_tab["1syso_float_var"].first ));
            /*invoke syso*/
            add_code("invokevirtual java/io/PrintStream/println(F)V");
        }
    }
    ;
```

TESTING DEMO

In order to test any java program, put the test file in **tests** folder and modify its name in **runfile** file.

Here are screen shots of running the program, and screen shots of the main rules added:

```
adel@adel-Inspiron-7577:~/Desktop/phase3$ make -f runfile compile_and_run
flex scanner.l
bison -y -d syntax.y
syntax.y: warning: 2 shift/reduce conflicts [-Wconflicts-sr]
g++ -std=c++11 lex.yy.c y.tab.c
./a.out tests/test7




java -jar ./jasmin-1.1/jasmin.jar output.j
Generated: test.class
java test
0
1
2
3
4
5
6
7
8
9
```

```
int x;
int y;
for ( x = 0 ; x < 10 ; x = x + 1;)
{
        System.out.println(x);
}
```

TESTING THE GENERATED BYTECODE

Here is how we tested that the generated bytecode (java bytecode assembler, Jasmin, ... etc.):

Open | bytecode_inst.h ~/Desktop/phase3 | Save

```
#ifndef _____BYTE_CODE_INSTRUCTIONS
#define _____BYTE_CODE_INSTRUCTIONS

#include <map>
using namespace std;

/* list of mnemonics that corresponds to specific operators */
map<string,string> inst_list = {
        /* arithmetic operations */
        {"+", "add"},
        {"-", "sub"},
        {"/", "div"},
        {"*", "mul"},
        {"|", "or"},
        {"&", "and"},
        {"%", "rem"},

        /* relational op */
        {"==", "if_icmpeq"},
        {"<=", "if_icmple"},
        {">=", "if_icmpge"},
        {"!=", "if_icmpne"},
        {">",  "if_icmpgt"},
        {"<",  "if_icmplt"}
};

#endif
```

C/ObjC Header | Tab Width: 8 | Ln 10, Col 22 | INS

THANK YOU