University *of* Alexandria
Faculty of Engineering
Computer and Systems Engineering Department

Compilers

Phase #3

Java Byte Code Generation

Names:

Ahmed Ibrahim Hafez	(04)
Ahmed El-Sayed Mahmoud	(05)
Amr Gamal Mohamed	(45)
Mohamed El-Sayed Mohamed	(54)

• ***Data Structures***

The used build-in data structures are: Vector, Map, Union

- typedef enum {INT_T, FLOAT_T, BOOL_T, VOID_T, ERROR_T} type_enum;
  It contains the primitive types that is allowed.
- extern int lineCounter;
  It keep track of number of code lines reached to be used later for example if there is an error it will till on which line is this error.
- map<string,string> inst_list;
  It contains the instructions allowed with its string that will be written in the byte code.
- map<string, pair<int,type_enum> > symbTab;
  It contains the variables declared and its address and type.
- vector<string> codeList;
  It contains Byte code then it's written once parser finish.
- union{List of Attributes}
  It contains all the attributes used in the Semantic Actions. Then we can give any terminal or nonterminal we need the attribute we want by using the syntax '%token<attribute> token_name' for terminals and '%type<attribute> rule_name' for nonterminals.

• ***Algorithms & techniques***

- **Syntax.y**
  It is the new parser written in bison convert a context-free grammar and semantics rules into a parse tree. Then execute the semantic rules to generate the java byte code.
- **Lex.l**
  It can do the rule of lexical analyzer i.e parse code to get tokens.

- The algorithm of Flow-Of-Control Translation

1- <u>If – Else:</u>

$S \rightarrow$ **if** E **then** S_1 $\quad$ { $E.true := newlabel;$
$E.false := S.next;$
$S_1.next := S.next;$
$S.code := E.code \,\|\, gen(E.true\ ':')\ S_1.code$ }

$S \rightarrow$ **if** E **then** S_1 **else** S_2 $\quad$ { $E.true := newlabel;$
$E.false := newlabel;$
$S_1.next := S.next;$
$S_2.next := S.next;$
$S.code := E.code \,\|$
$\quad gen(E.true\ ':')\ S_1.code \,\|$
$\quad gen(\texttt{'goto'}\ S.next) \,\|$
$\quad gen(E.false\ ':')\ S_2.code$ }

2- <u>Boolean Expression:</u>

we generate the appropriate branch code using the inherited true and false attributes.

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \,\|\, label(B_1.false) \,\|\, B_2.code$
$B \rightarrow B_1\ \&\&\ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \,\|\, label(B_1.true) \,\|\, B_2.code$
$B \rightarrow\ !\ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \,\|\, E_2.code$ $\|\, gen(\texttt{'if'}\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ \texttt{'goto'}\ B.true)$ $\|\, gen(\texttt{'goto'}\ B.false)$
$B \rightarrow$ **true**	$B.code = gen(\texttt{'goto'}\ B.true)$
$B \rightarrow$ **false**	$B.code = gen(\texttt{'goto'}\ B.false)$

3- <u>While:</u>

```
S → while E do S₁ { S.begin := newlabel;
                    E.true := newlabel;
                    E.false := S.next;
                    S₁ .next := S.begin;
                    S.code :=  gen (S.begin ':' )  || E.code  ||
                               gen (E.true ':'  ) ||  S₁ .code ||
                                   gen ('goto'  S.begin) }
```

- ***Comments about tools***
  - Flex: Takes the input code and produces Stream of tokens according to the given Lexical Rules. The Resultant Stream of tokens is passed to the Bison tool.
  - Bison: Takes the stream of tokens results from the Flex output and parses the token according to the given Context Free Grammar and Semantic Actions to generate the required Java Byte Code.
  - Jasmin: Takes the generated Byte Code and runs it using the Java Virtual Machine (JVM) to test its result.
- ***Functions Explanations***

void generateHeader(void);

- Generate header for class to be able to compile the code.

void generateFooter(void);

- Generate footer for class to be able to compile the code.

void genCode(vector<int> *list, int num);

- It takes the next list of any nonterminal and adds labels to the goto of each next if more than one next exists to the nonterminal.

void defineVar(string name, int type);

- It checks whether the variable already exists in the symbol table or not and if not it add the suitable code to codeList and inserts the symbol into the symbol table.

string generateLabel();

- It generates a new label to be used in the program.

• ***Assumptions & Justification***

1) The Variable can be declared only as global variables (life scope variables not considered).
2) All if statements are followed with else statements. As the grammar only had if-else not if alone.
3) There is no casting to variables yet (assignments must be of the same type).
4) Declaration and assignment of a variable must be done on two stages.
5) ++ operator not supported.