Compilers - Phase III Report

1. Enas Gaber (23)
2. Dina Hossam (30)
3. Monica Salama (73)
4. Mostafa Gabriel (67)

Objective

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

Description:

Generated bytecode must follow Standard bytecode instructions defined in Java VirtualMachine Specification.

Data structures:

- struct {
  ```
  int ival;
  float fval;
  int address;
  int start;
  string* val;
  string* type;
  vector<string> *code;
  } node;
  ```

Struct node used to identify int value or float value , get address, start line, text value of ID, string type of the ID

And vector<string> to store the code of this node.

- map<string, pair<string, int> > myMap
  Mapping each string IDs and constants to its type and address
- map<float, int> fMap;

Algorithms and techniques:

- %token <node> INT FLOAT IF ELSE WHILE ID NUM RELOP ASSIGN SEMICOLON COMMA OP CL OPC CLC PLUS MINUS MULT DIV
- %type <node> method_body statement_list statement declaration primitive_type if else while assignment
- Global variable PC

1. build the lexical rule file to identify the tokens
2. write semantic rules beside CFG in file.y that generate the java byteCode
3. build the parse tree using bison to output the java byteCode.

Tools:

Flex and Bison : are utilities that help you write very fast parsers.

They implement Look-Ahead-Left-Right parsing of non-ambiguous context-free.

First, Flex and Bison will generate a parser that is guaranteed to be faster than anything you could write manually in a reasonable amount of time.

Second, updating and fixing Flex and Bison source files is a lot easier than updating and fixing custom parser code.

Third, Flex and Bison have mechanisms for error handling and recovery.

Functions:

```
string itos(int i) {
   stringstream s;
    s << i;
    return s.str();
}
```
This function used to convert int to string.

```
void printVector(vector<string>* v) {
        for (int i = 0; i < v->size(); i++)
                cout << (*v)[i] << endl;
}
```
This function used to print vector contain bytecode.

```
string iRelopCode(string* s) {
        if(s->compare("<") == 0)
                return "if_icmpge ";
        if(s->compare(">") == 0)
                return "if_icmple ";
        if(s->compare(">=") == 0)
                return "if_icmplt ";
        if(s->compare("<=") == 0)
                return "if_icmpgt ";
        if(s->compare("==") == 0)
                return "if_acmpne ";
        if(s->compare("!=") == 0)
                return "if_acmpeq ";
}
```
This function used to check the relop and return the right instruction.

```
string fRelopCode(string* s) {
        if(s->compare("<") == 0)
                return "fcmpg\nifge ";
        if(s->compare(">") == 0)
                return "fcmpl\nifle ";
        if(s->compare(">=") == 0)
                return "fcmpl\niflt ";
        if(s->compare("<=") == 0)
                return "fcmpg\nifgt ";
        if(s->compare("==") == 0)
                return "fcmpl\nifne ";
        if(s->compare("!=") == 0)
                return "fcmpl\nifeq ";
        return "BS";
}
```
This function used to check the relop and return the right instruction (for float).

Assumptions:

- Change nonterminal sign:+|- into sign: PLUS | MINus
- Divide addop into "+"   { return PLUS; }
  "-"   { return MINUS; }

- Divide addop into "*"   { return MULT; }
  "/"   { return DIV; }

- Add new Rule in the CFG -> else: ELSE