Compiler
Phase 3
Java Byte Code Generation

Names :

Ahmed Ezzat Elmaghawry	(11)
Arsanous Essa Attia	(18)
Saeed Hamdy	(30)
Marwan Morsy	(69)

Objective :

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

Description:

Generated byte-code must follow Standard byte-code instructions defined in Java Virtual Machine Specification

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

Introduction :

This phase is divided into several parts which are:

1. Lexical Analyzer (lex.l) : in this phase we tokenize the input and define the meaning of every set of characters in the input code.

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.

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.

Data Structures used :

*** Vector :**

*** Enum :**

In Bison, union directive specify 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 <val> INT_CONST , then we have INT_CONST that has a semantic value of int .

Algorithms used :

Functionality of the project :

1. **Scanning**

   In this step the lexical analyzer for the proposed language is constructed. We used a scanner generator can be used lex of Flex for C/C++ .

   2.Parsing

   In this step parser for the proposed language is constructed. The parser has two main goals:

   •to check whether the input is a syntactically correct program, and
   •to generate an abstract syntax tree (AST) that records all important information about the program (the intermediate representation); to construct AST, some actions must to be added to the parser.

   A parser generator can be used (yacc or Bison for C/C++)

3. **Semantic analysis**

A two-pass static-semantic analyzer for the programs represented as abstract-syntax trees has to be constructed:

- the name analysis method, and
- type checking method.

The semantic analyzer will traverse the tree constructed in parsing phase to build a table of the symbols contained in the programs and to decorate AST with semantic information.

Name analysis perform several tasks, like: building symbol table, finding bad declarations, multiply declared names and uses of undeclared names, etc. They are used to generate useful error messages in this phase.

The job of the type checker is to determine the type of every expression in the abstract syntax tree and verify that the the type of each expression is appropriate for its context.

4. **Code generation**

Finally, the abstract syntax tree will be traversed again to perform final code generation. The code will be generated for a specified assembly language. The target language can be ASM (the assembly language for Intel processors) or Jasmin assembly language (for Java virtual machines).

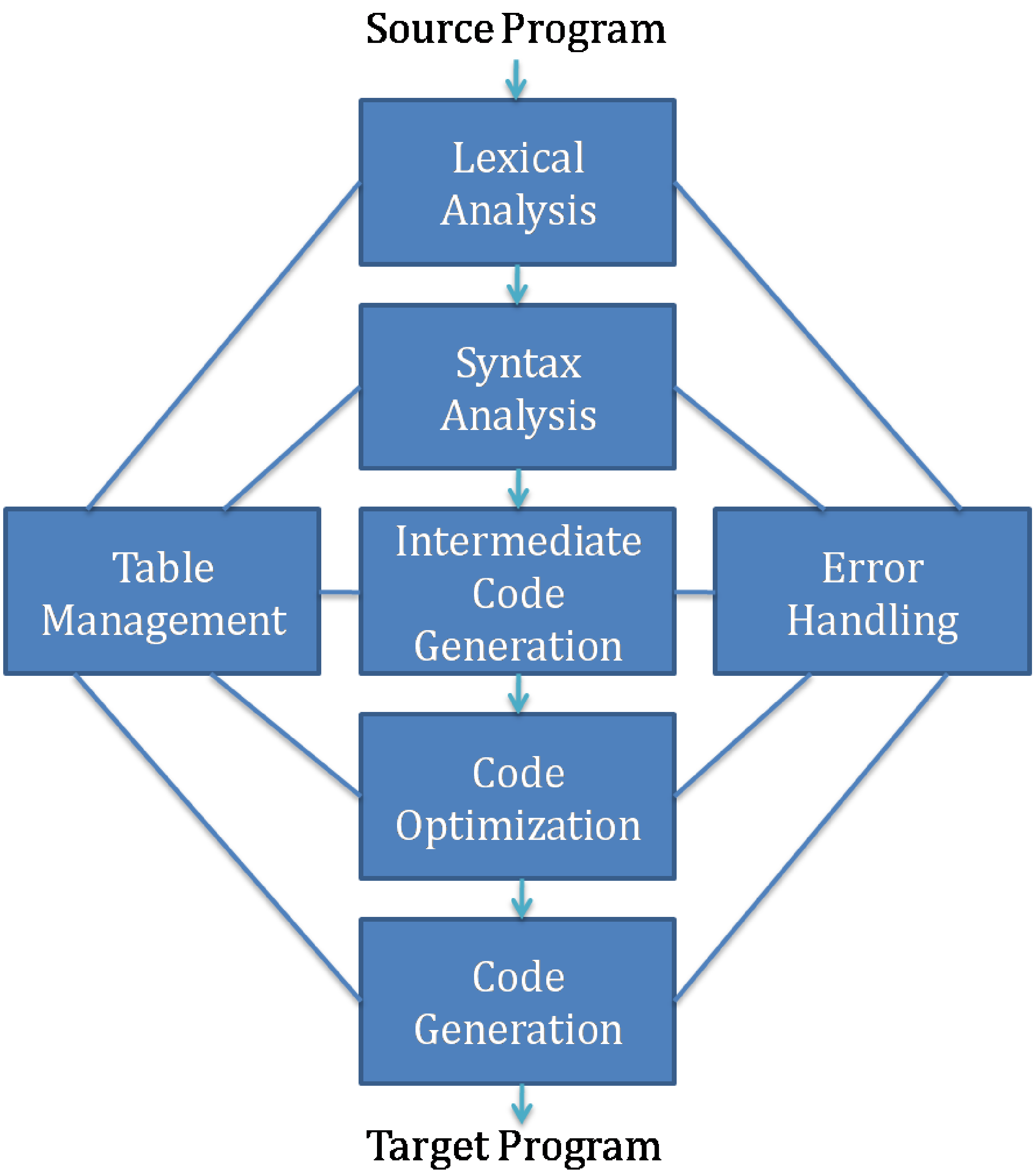

Fig. Phases of a Compiler

Comments about used tools :

- **Lex**

It is a tool to generate a Lex program then run this program on any input program to generate the lexical output and symbol table

after enter in command line : lex token.l
it will generate a .c file lex.yy.c file

- **Jasmin** :

  It is an assembly language for Java byte code. The assembler reads as input a Jasmin assembly program and produces as output a Java class file ready to be run on a JVM.

- **Bison :**

  It is a parser generator that reads a specification of a context free grammar, warns about any parsing ambiguities, and generates a parser which reads sequences of tokens and decides whether the sequence has correct syntax according to the grammar or not.

* Any assumptions made and their justification :

- There is unreasonable error at adding %code to .y file so we made an assumption to run the project :

```
1- flex tokens.l
2- bison -y -d Bisi.y
3- Add to y.tab.c :
            #include<vector>
            using namespace std;
4- g++ -std=c++11 lex.yy.c y.tab.c
5- ./a.out code.txt
```

Reference : http://aquamentus.com/flex_bison.html