COMPILER DESIGN PROJECT

Visualization of Phases of Compiler in JAVA

Submitted by:

Subham Gupta (13103456)
Aryan Jadon (13103734)
Burhanuddhin (13103735)
Prachi Tiwari (13103742)

Submitted to:

Ms. Aayushee Gupta

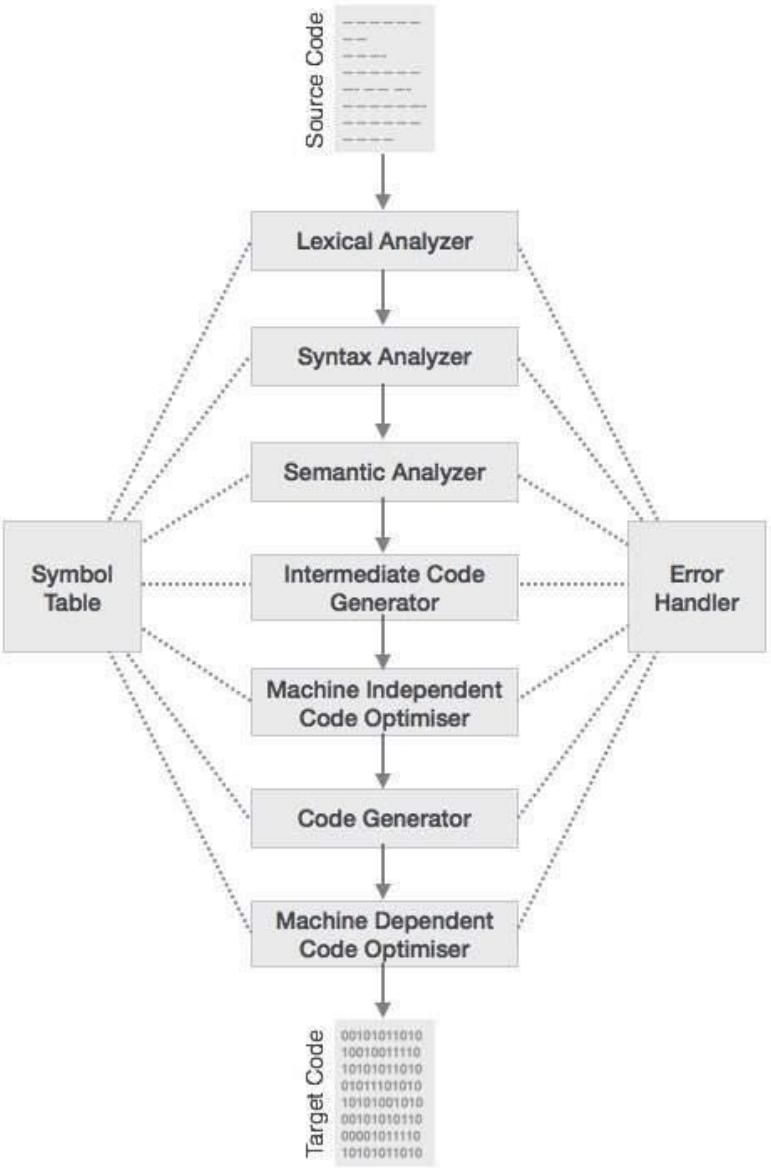
Batch- B10

ABSTRACT

A compiler translates the code written in one language to some other language without changing the meaning of the program. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space.

Compiler design principles provide an in-depth view of translation and optimization process. Compiler design covers basic translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end.

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.



Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

```
<token-name, attribute-value>
```

Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

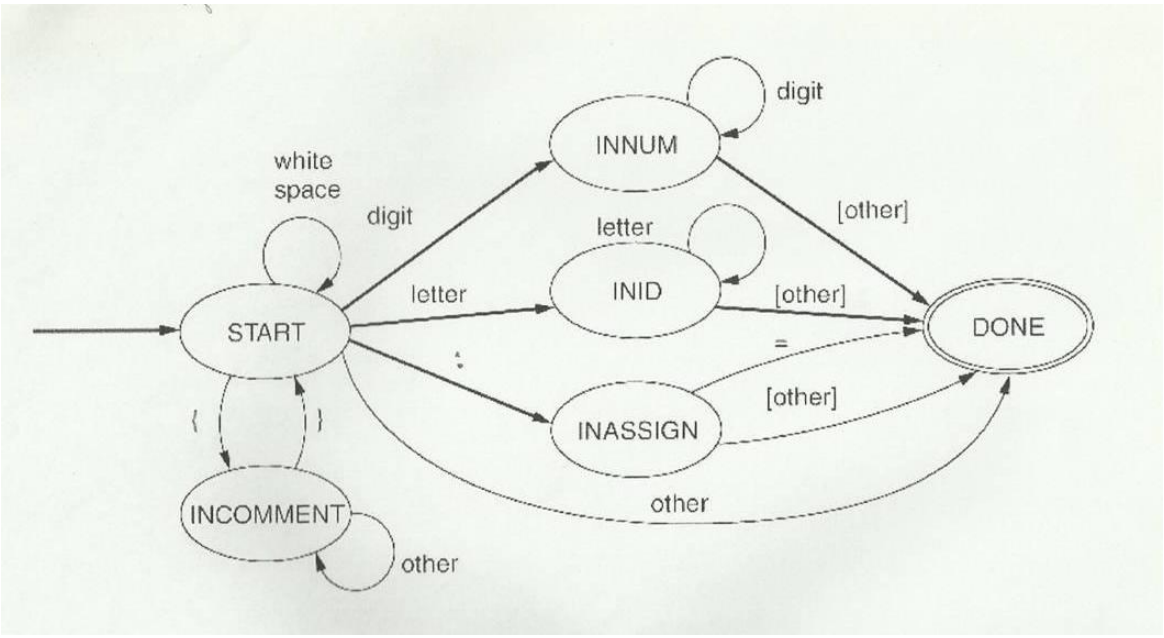
Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

DFA USED



GRAMMAR USED

program → stmt-seq
stmt-seq → stmt stmt-seq`
stmt-seq` → ; stmt-seq`
stmt-seq` → ε
stmt → if-stmt
stmt → repeat-stmt
stmt → assign-stmt
stmt → read-stmt
stmt → write-stmt
if-stmt → if exp then stmt-seq
if-stmt` if-stmt` → end

if-stmt` → else stmt-seq end
repeat-stmt → repeat stmt-seq until exp
assign-stmt → identifier := exp
read-stmt → read identifier
write-stmt → write exp
exp → simple-exp exp`
exp` → comparison-op simple-exp exp` → ε
comparison-op → <
comparison-op → =
simple-exp → term simple-exp`
simple-exp` → addop term
simple-exp`
simple-exp` → ε
addop → +
addop → -
term → factor term`
term` → mulop factor term`
term` → ε
mulop → * mulop → /
factor → (exp)
factor → number
factor → identifier

First Set of Non-Terminals:

first(program) = {if, repeat, identifier, read, write} first(stmt-seq) = {if, repeat, identifier, read, write}

first(stmt-seq`) = {;, ε}

first(stmt) = {if, repeat, identifier, read, write} first(if-stmt) = {if}

first(if-stmt`) = {end, else} first(repeat-stmt) = {repeat} first(assign-stmt) = {identifier} first(read-stmt) = {read} first(write-stmt) = {write}

first(exp) = {(, number, identifier} first(exp`) = {<, =, ε}

first(comparison-op) = {<, =}

first(simple-exp) = {(, number, identifier} first(simple-exp`) = {+, -, ε}

first(addop) = {+, -}

first(term) = {(, number, identifier} first(term`) = {*, /, ε}

first(mulop) = {*, /}

first(factor) = {(, number, identifier}

Follow Set of Non-Terminals:

follow(program) = {$} follow(stmt-seq) = {$, end, else, until} follow(stmt-seq`) = {$, end, else, until} follow(stmt) = {;, $, end, else, until} follow(if-stmt) = {;, $, end, else, until} follow(if-stmt`) = {;, $, end, else, until}

follow(repeat-stmt) = {;, $, end, else, until} follow(assign-stmt) = {;, $, end, else, until} follow(read-stmt) = {;, $, end, else, until} follow(write-stmt) = {;, $, end, else, until} follow(exp) = {then, ;, $, end, else, until,)} follow(exp`) = {then, ;, $, end, else, until,)} follow(comparison-op) = {(, number, identifier}

follow(simple-exp) = {<, =, then, ;, $, end, else, until,)} follow(simple-exp`) = {<, =, then, ;, $, end, else, until,)} follow(addop) = {(, number, identifier}

follow(term) = {+, -, <, =, then, ;, $, end, else, until,)} follow(term`) = {+, -, <, =, then, ;, $, end, else, until,)} follow(mulop) = {(, number, identifier}

follow(factor) = {*, /, +, -, <, =, then, ;, $, end, else, until,)}

Implementation:

I. Tokens are divided as follows:

0. Number
1. ID
2. Left Parenthesis
3. Right Parenthesis
4. Plus
5. Minus
6. Divide
7. Multiplication
8. Equal
9. Assignment
10. Error
11. *SEMICOLON*

12. *SMALLERTHAN*
13. GREATER THAN
20. IF
21. *THEN*
22. *ELSE*
23. END
24. REPEAT
25. UNTIL
26. READ
27. WRITE
Then it prints “Scanning is complete” if the scanning is done successfully.

II. The Parser receives the tokens from the scanner using the function *getToken();* reading from the file *“test.txt”*.
According to the grammatical rules in EBNF:
program -> stmt-sequence
stmt-sequence -> stmt {; stmt}
stmt -> if-stmt |repeat-stmt | assign-stmt | read-stmt | write-stmt
if-stmt -> if exp then stmt-sequence [else stmt-sequence] end
repeat-stmt -> repeat stmt-sequence until exp
assign-stmt -> identifier := exp
read-stmt -> read identifier
write-stmt -> write exp
exp -> simple-exp [comparison-op simple-exp]
comparison-op -> < | =
simple-exp -> term {addop term}
addop -> +| -
term -> factor {mulop factor}
mulop -> *| /
factor -> (exp) | number | identifier
The Parser checks whether the input text matches the grammatical rules or not.
If yes, it prints “Parsing complete”.
If not, it prints “Parsing error” with hints about the error place of the received token.
It also prints the matched token values.

III. Created an HTML Page that takes input as a source code and on pressing ‘Start Button’ generates Visualization Of Token generated. This is done using Javascript and d3(Document Data Viewer Library). The user is also given a feature to be able to control the speed of the scanning Phase of the Compiler.