Javier

SER-502

Spring 2017

Team 9

Doug Carroll, Jeb Johnson, Isidro Perez, Jason Rice

Design

The language and compiler will be implemented using Java 1.8. We decided to use Java as a base for the compiler so that the language is platform independent.

Lexical Analysis and Parsing Technique

Parsing begins with lexical analysis. Lexical analysis will be done from scratch in Java 1.8. The lexical analysis portion of the compiler will read the code from the users file. Once read in, the code is stripped into tokens based on the rules we set in the grammar/grammar.json file. Each token will be an object of Token type, containing a type value and the actual token itself. A list of these tokens is created, omitting any commented out tokens. This token list will then be sent to the parser.

The parser will begin to scan the token list, checking if they are valid based on our grammar rules and semantic rules defined for the language. This check for validity will be done by adding each token to the parse tree based on these rules. During this step, the parser will also check if the lexical analysis picked up any errors by checking token type for an error value. If any errors are found during this step, the user will be notified of the specific error via terminal output and compilation will terminate.

The parse tree will then be broken down statement by statement and translated to intermediate code.

Intermediate Code

The intermediate code will be our own assembly-like code. This code will be similar to the MIPS assembly language. The largest difference between the two is that our version will hold the actual variables with their name. This will make it easier for parsing/runtime. We are able to do this since we do not have to deal with registers. The parser will put the code into PEMDAS order for use by the runtime. An example of this intermediate code is below:

Actual Code Example:

```
x = 5 + 2 - 3
if(x > 3){
    print x
}
```

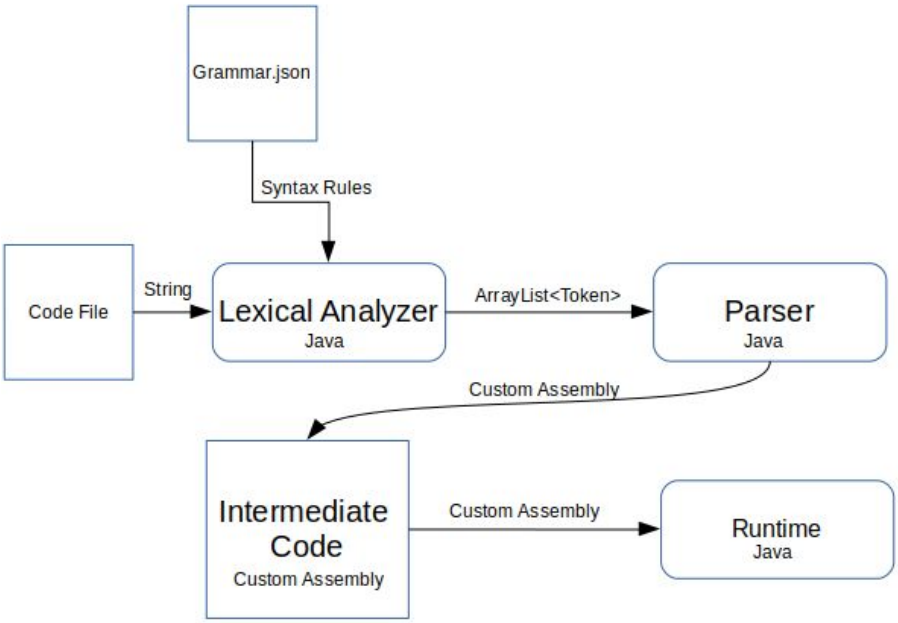
Intermediate Code:

```
add,x,5,2    # Addition
sub,x,x,3    # Subtraction
gt,t1,x,3    # Greater than
if,t1,label  # If statement
print,x      # print statement
label        # label to jump to if false
```

The first word is the instruction or opcode, the next value is an assignment for that instruction and the next two values are the two values associated with the instruction. The if statement will jump to our label if t1 is false. The value 4 will be printed in the above example.

Interpreter Design

The runtime/Interpreter is implemented by traversing line by line through the intermediate code. Each new line character denotes a separate instruction to be run. Each of these instructions will be loaded into an array and run sequentially. Variables taken from the intermediate code will be stored in the symbol table via a hashmap data structure. Before the actual running of the program, the entire intermediate code will be traversed and labels will be added to the symbol table along with their associated line numbers. During execution, when a jump is encountered, the label will be used to jump to that line number in the intermediate code array. Instructions will be translated/executed immediately when they are encountered by the runtime. The runtime will be made from scratch using Java.



Grammar

- Program → Statement-Lst
- Number → Digit | Digit Number
- Digit → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
- Letter → ‘a’..’z’ | ‘A’..’Z’
- Identifier → Letter | Letter Number | Letter Identifier
- Assignment → Identifier “=” Expr
- If → “if” Condition “{“ Statement-Lst “}” | If Else
- Else → “else” “{“ Statement-Lst “}”
- Statement-Lst → Statement | Statement Statement-Lst
- Condition → Expr “==” Expr | Expr “>=” Expr | Expr “<=” Expr | Expr “>” Expr | Expr “<” Expr | Condition “and” Condition | Condition “or” Condition | “true” | “false”
- Statement → “” | Assignment | If | Loop
- Iterator → Expr “,” Expr
- Loop → “loop” Loop-Assignment “{“ Statement-Lst “}”
- Loop-Assignment → Identifier “=” Iterator
- Expr → Expr “+” Expr | Expr “-” Expr | Expr “*” Expr | Expr “/” Expr | Expr “%” Expr | “(“ Expr “)” | Number