

NIHAO –Milestone 2

TEAM MEMBERS:

Jiuxu Chen

Satish Bhambri

Vibhuti Tripathi

Prashanth Radhakrishnan

The name of our compiler and corresponding grammar 'Nihao' comes from the Chinese word Nihao, meaning hello. The inspiration behind the name comes from one of our team member Jiuxu.

Designing the compiler consists of significant stages and we have fairly distributed the work of every stage between our different team members.

The first stage involves defining the structure of the language and providing meaning to it, also known as grammar.

Our language satisfies following constructs:

1. The programs written in this language can incorporate following primitive operators:
 - a. '+' for addition
 - b. '-' for subtraction
 - c. '*' for multiplication
 - d. '/' for the division

Bitwise operators are not supported in our language.

2. Nihao supports one numeric type that is **int**. It doesn't support the other numeric types of float and double.
3. Nihao supports the decision control construct of 'if-then-else'
 - a. The language identifies the construct occurrence in the program by the identifiers of 'start' and 'end'.
 - b. It doesn't support any other decision control construct such as Switch case scenario.
4. Nihao supports 'while' looping construct for iterative execution and recognizes the start and end of the loop using the identifiers 'do' and 'od'.
5. For defining our grammar we've used Extended Backus-Naur Form of Context Free Grammar. We used EBNF to avoid recursive descent parsing. Left recursion in productions generates infinitely recursive code.
Eg: BNF : $N \rightarrow ND \mid D$
EBNF: $N \rightarrow D\{D\}$

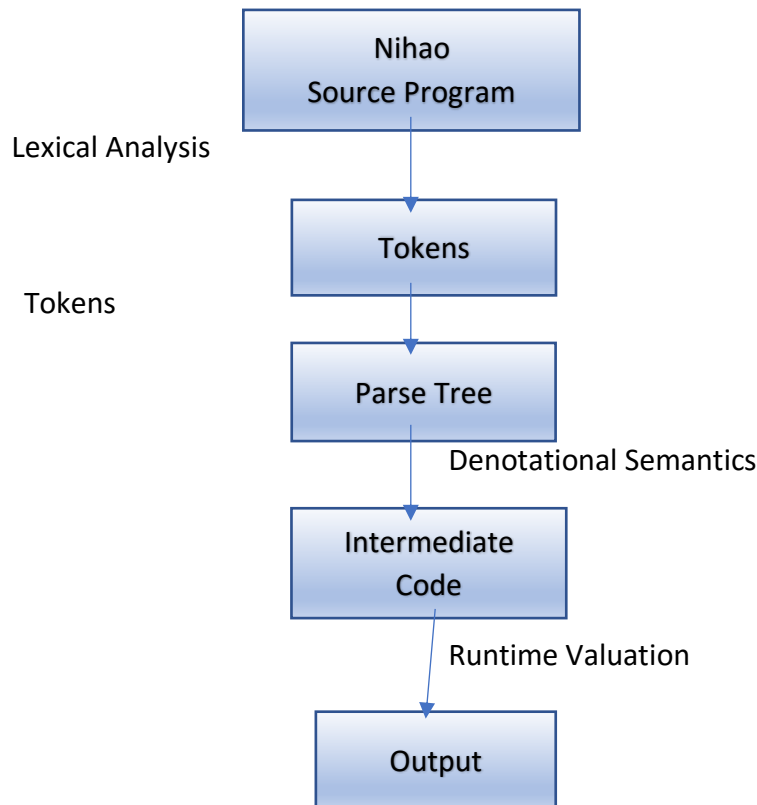
6. The Lexical Structure of our Language consists of following:

Reserved Keywords: while, do, od, if, start, end

Constants = 0,1,2,3,4,5,6,7,8,9

Special Symbols = '+', '-', '/', '*', '>', '<', '<=', '>=', '==', '%'

Identifiers = 'a', 'b' 'z'



The syntax of our grammar supports the Data Type **Int**, Decision Control Statement **If–then and Else**, Assignment Operator **'='**, **While** Loop for iterative execution. The syntax does not support curly braces to mark the beginning and end of the loops, instead we have used the keywords **'do'** and **'od'** for while loop and **'start'** & **'end'** for if–then else construct which mark the beginning and end of it.

Our language supports addition, subtraction, multiplication and division as the arithmetic operations and supports the order of precedence.

We've defined the syntax of our language to be a program which consists of a **statement set** which might terminate on one statement or further have a statement set.

We've implemented recursion in our grammar for this purpose.

The statement further supports an assignment statement, an iterative statement or a decision control statement.

Nihao follows an imperative language paradigm, supporting sequential execution of instructions. It doesn't support any method calls or function calls.

We aim to generate intermediate code language at the level of a byte-code language using a stack machine model, or a model in which instructions have an opcode and one or two operands.

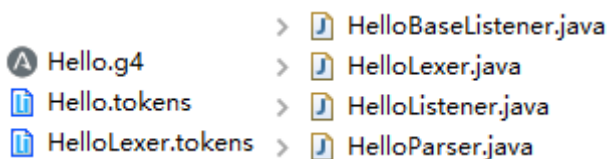
The iterative statement, While loop consists of an expression, the keyword "do" and "od" which mark the beginning and end of the loop and similarly for If loop defined with the keyword "start" and "end".

For lexical Analysis i.e for splitting our grammar into token we've used ANTLR(Another Tool for Language Recognition) which takes the grammar .g4 file as the input and produces two classes lexer and parser. The lexer class converts the grammar into a stream of tokens which are then converted into a parse tree via parser.

We've used LL(Top Down) Parsing Technique for our compiler which involves finding the terminals to find the beginning of the production rules. Once the start terminal is known we compare the start sets of the productions against the input to determine the already used productions. The tool used for lexical analysis uses a top down approach, generating recursive descent parsers called LL(*) that allow semantic predicates.

We have the ANTLR Grammar which could be identified by Antlr4.7 and we will apply it to Antlr4.7 in Eclipse Neon so that it can automatically generate the parse-tree.

First, Antlr4.7 will identify the .g4 file so that Eclipse will compile it and generate .tokens and .java files.



The screenshot shows the Eclipse IDE's project explorer with a project named 'Hello'. It contains the following files and folders:

- Folder:** Hello.g4
 - File:** HelloBaseListener.java
 - File:** HelloLexer.java
- File:** Hello.tokens
 - File:** HelloListener.java
- File:** HelloLexer.tokens
 - File:** HelloParser.java

Then, we are going to create a java program to get all those components work.

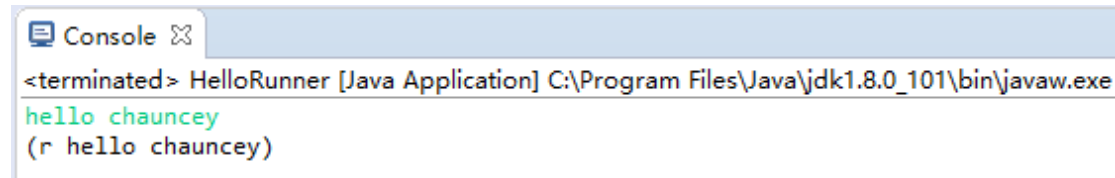
```

1 import org.antlr.v4.runtime.*;
2 import org.antlr.v4.runtime.tree.*;
3 public class HelloRunner
4 {
5     public static void main( String[] args) throws Exception
6     {
7
8         @SuppressWarnings("deprecation")
9         ANTLRInputStream input = new ANTLRInputStream( System.in);
10        CharStream cs = CharStreams.fromStream(System.in);
11
12        HelloLexer lexer = new HelloLexer(input);
13        HelloLexer lexer = new HelloLexer(cs);
14
15        CommonTokenStream tokens = new CommonTokenStream(lexer);
16
17        HelloParser parser = new HelloParser(tokens);
18        ParseTree tree = parser.r(); // begin parsing at rule 'r'
19        System.out.println(tree.toStringTree(parser)); // print LISP-style tree
20    }
21 }

```

If the Grammar is wrong, it will not be compiled. However, in case the grammar is not properly defined but still can be compiled, we test it with some input to make sure that it is generating the expected parse-tree.

The final step is to test the grammar. Since we will do the Antlr4.7-parsing using Eclipse Neon, we can directly give the input and output parse-tree in the console. For example:



```

<terminated> HelloRunner [Java Application] C:\Program Files\Java\jdk1.8.0_101\bin\javaw.exe
hello chauncey
(r hello chauncey)

```

(r hello chauncey) is the output, the parse-tree. We are going to check if it properly matches the grammar we defined.

```

grammar Hello;
r : 'hello' ID ;

ID : [a-z]+ ;

WS : [ \t\r\n]+ -> skip ;

```

“r” means the rule. The rule constrains that the input should start with “hello”, followed by “ID” which means the identifiers that should be a set of lower-case alphabet. So, in this case, the whole process is executed successfully.

In our project, we will translate the whole EBNF grammar into ANTLR grammar and test it in the approach mentioned above repeatedly to make sure everything works.

Semantic Analysis:

For semantic analysis, we'll be using Denotational Semantics. The semantic analysis of a program is essential to provide meaning to it and denotational semantics provides meaning to the program using mathematical objects such as integers, truth values and functions. Also, Denotational semantic representations can be easily converted into the working programs.

Semantic functions map objects of the syntax of our language with their semantics. Constructs of the subject language—namely elements of the syntactic domains—are mapped into the semantic domains. We plan to use valuation function to map the program to its semantics.

A function describes semantics by associating semantic values to syntactically correct constructs, for instance a function that maps an integer arithmetic expression to its value, which we could call the Val function:

$\text{Val} : \text{Expression} \rightarrow \text{Integer}$

Since Val maps the syntactic construct to the semantic value 14, this is the origin of the name denotational semantics.

The denotational definition of Nihao consists of three parts:

- A definition of the **syntactic domains** on which the semantic functions act.
 - Syntactic domain here is represented by the grammar defined for Nihao, which only allows certain syntax to be acceptable.
- A definition of the **semantic domains** consisting of the values of the semantic functions.
 - Semantic domain here includes the mathematical functions and operators.
- A definition of the semantic functions themselves also called valuation functions.

We will generate the intermediate code using symbol table, used to store the state of the program with the execution of every instruction.

RunTime Environment:

Runtime environment is a state which includes software libraries, environment variables, etc., to provide services to the processes running in the system. Once we've done lexical, syntactic and semantic analysis we plan to design the run time environment so that the instructions can be executed and the procedure calls need to be assigned memory at run time and need to be deallocated when not in use.

We are using java to build the runtime environment. Run time would correspond to the JVM, where we would run the intermediate code generated.

In order to keep a track of the procedure calls and memory allocation-deallocation we plan to use Symbol Table as our data structure. The symbol table will be responsible for storing information about the identifiers, variables etc, ie the state of the program with the execution of every instruction.

We plan to implement our symbol table in the form of a hash table which will map the entity and its data type. Eg our compiler supports int data type so for instance [int score] will be stored as <int, store>.

We also plan to implement the following functionalities in the hash table :

- 1) Insert: so that a new identifier can be inserted everytime the lexer come across it.
- 2) Search: so that we can determine if the symbol exists in the table or not.

EBNF Grammar:

Program = statement {statement}

variable = char {char}

character ::= 'a' | 'b' | 'c' | . . . | 'z'

statement = assignment | if_statement | while_statement

assignment = [datatype] variable '=' expression

if_statement = 'if' '(' (expression) ')' 'start' Program ['else' Program] 'end'

while_statement = 'while' '(' (expression) ')' block

block = 'do' Program 'od'

datatype = 'int'

expression = expression2 {(">" | ">=" | "<" | "<=" | "==") expression2}

expression2 = term {(">" | "<") term}

term = factor {("*" | "/" | "%") factor}

factor = constant | variable | "(" expression ")"

variable = char {char}

char = "a" | "b" | "." | "z"

constant = digit {digit}

digit = "0" | "1" | "." | "9"