Bilkent University

Department of Computer Engineering

Spring 2025
Section 1
Instructor: H. Altay Güvenir

CS 315 Project 1

Lexical Analyser for a Programming Language for a Real Numbers

Group Members:
Emine İrem Esendemir - 22202103
Irmak İmdat - 22201570
Sude Ergün - 22203822

LADYBUG

1.The complete BNF description

- **Program**

<program> ::= int_main **{** <stmts> **}** <function_decs>

<function_decs> ::= <function_dec> <comments> <function_decs> | <function_dec>

<stmts> ::= <stmt> <comments> <stmts> | <stmt>

<stmt> ::= ε | <dec_stmt>; | <assign_stmt>; | <condition_stmt> | <loop_stmt> |

<return_stmt>; | <input_stmt>; |<output_stmt>;

- **Base Declarations**

<letter> ::= **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z** | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** |**Y** | **Z**

<special_char> ::= | **!** | **$** | **%** | **&** | **|** | ***** | **+** | **,** | **-** | **.** | **/** | **:** | **;** | **<** | **=** | **>** | **?** | **@** | **[** | **** | **]** | **^** | **_** | **{** | **}** | **~** | ” | ’ | \ |\t

<newline> ::= \n

<digit> ::= **0** | **1** | **2** | 3 | ... | 9

<sign> ::= **+** | **-**

<multiplicative_op> ::= ***** | **/** | %

<expon_op> ::= **^**

<point> ::= **.**

<relational_op> ::= **>** | **<** | **>=** | **<=** | **==** | **!=**

<type> ::= **int** | **float**

<logical_op> ::= **&&** | **||**

<not> ::= **!**

- **Constant real values**

<constant> ::= <integer> | <integer> <point> <integer>
<integer> ::=<digit> | <integer> <digit>

- **Variables**

<alphanumeric> ::= <letter> | <digit>
<variable> ::= <letter> | <variable> <alphanumeric>
<all_char> ::= <alphanumeric> | <special_char>
<str> ::= ε | <all_char> | <str> <all_char>
<str_with_newline> ::= ε | <all_char> | <str_with_newline> <all_char>
| <str_with_newline> <newline>

- **Expressions**

<expr> ::= <or_expr>
<or_expr>::= <and_expr> | <or_expr> **||** <and_expr>
<and_expr> ::= <relational_expr> | <and_expr> **&&** <relational_expr>
<relational_expr> ::= <arithmetic_expr>
| <arithmetic_expr> <relational_op> <arithmetic_expr>
<arithmetic_expr> ::= <term> | <arithmetic_expr> <sign> <term>
| <sign> <arithmetic_expr>
<term> ::= <factor> | <term> <multiplicative_op> <factor>
<factor> ::= <base> | <base> <expon_op> <factor> | <not> <factor>
<base> ::= <operand> | **(** <expr> **)**
<operand> ::= <variable> | <constant> | <function_call> | <arr_variable>

- **Declarations and Assignments**

<dec_stmt> ::= <type> <dec_parts> | **const** <type> <dec_parts>
<dec_parts> ::= <dec_part> | <dec_part> **,** <dec_parts>
<dec_part> ::= <variable> | <assign_stmt> | <array_dec_part>
<array_dec_part> ::= <variable> **[] = {** <exprs> **}** | <variable> <arr_size_decs>

| <variable> <arr_size_decs> **= {** <multi_exprs> **}**

<arr_size_decs> ::= <arr_size_dec> <arr_size_decs> | <arr_size_dec>

<arr_size_dec> ::= **[** <integer> **]**

<arr_exp> ::= **{** <exprs>**}**

<arr_exps> ::= <arr_exp> | <arr_exps>**,** <arr_exp>

<multi_exprs> ::= <exprs> | arr_exps> | <multi_exprs_with_braces>

<multi_expr_with_braces> ::= **{** <multi_exprs> **}**

<multi_exprs_with_braces> ::= <multi_expr_with_braces>

| <multi_expr_with_braces> **,** <multi_exprs_with_braces>

<assign_stmt> ::= <variable> **=** <expr> | <arr_assign> | <variable>**++** | <variable>**--**

<arr_variable> ::= <variable> <arr_size_decs>

<arr_assign> ::= <arr_variable> **=** <expr>

<exprs> :: = <expr> | <expr> **,** <exprs>

<return_stmt> ::= **return** <expr>

- **Conditional statements**

<condition_stmt> ::= **if (** <expr> **) {** <stmts> **}**

| **if (** <expr> **) {** <stmts> **} else {** <stmts> **}**

| **if (** <expr> **) {** <stmts> **}** <else_if_stmts>

| **if (** <expr> **) {** <stmts> **}** <else_if_stmts> **else {** <stmts> **}**

<else_if_stmts> ::= **else if (** <expr> **) {** <stmts> **}**

| **else if (** <expr> **) {** <stmts> **}** <else_if_stmts>

- **Loops**

<loop_stmt> ::= **for (**<type> <variable> **=** <expr> **;** <expr> **;** <assign_stmt>**) {** <stmts> **}**

| **for (**<type> <variable> **in** <variable> **) {** <stmts> **}**

| **while (**<expr>**) {** <stmts> **}**

| **do {** <stmts> **} while (** <expr> **)**

| **for (**<assign_stmt> **;** <expr> **;** <assign_stmt>**) {** <stmts> **}**

- **Input/output statements**

<input_stmt> ::= <variable> **= read**()

<output_stmt> ::= **write(“**<str_with_newline>**”)** | **write(**<expr>**)**

- ## Function definitions

<function_dec> ::= **funct** <variable> **(** <parameters> **) {** <stmts> **}** | ε

<parameters> ::= ε | <type> <variable> | <parameters>, <type> <variable>

<function_call> ::= <variable> **(** <arg_list> **)**

<arg_list> ::= <exprs> | ε

- ## Comments

<comments> ::= <comment> | <comment> <comments>

<comment> ::= ε | <line_comment> | <multiline_comment>

<line_comment> ::= **#** <str> <newline>

<multiline_comment> ::= **##** <str_with_newline> **##**

2. Non-Terminals

- **Program**

<**program**>

This non-terminal represents the main structure of the code in the language which consist of a main function and additional function declarations.

<**function_decs**>

This non-terminal represents the function declarations in the language. It can consist of one or more functions along with their comments.

<**stmts**>

This non-terminal represents the statements in the language. Statements can consist of one or more statements along with their comments.

<**stmt**>

This non-terminal represents a single statement in the language. A statement can be empty, a declaration statement, an assignment statement, a condition statement, a loop statement, a return statement, an input statement, or an output statement; all of which are explained in the following definitions.

- **Base Declarations**

<**letter**>

This non-terminal represents any uppercase or lowercase English letter (A-Z, a-z).

<**special_char**>

This non-terminal represents special characters that can be used in the program, including spaces (\ , \t), punctuation marks, and various symbols.

<**newline**>

This non-terminal represents a newline character (\n).

<**digit**>

This non-terminal represents a single numerical digit from 0 to 9.

<**sign**>

This non-terminal represents a mathematical sign, either positive (+) or negative (-).

<**multiplicative_op**>

This non-terminal represents multiplicative operators used in arithmetic expressions, including multiplication (*), division (/), and modulus (%).

<**expon_op**>

This non-terminal represents the exponentiation operator (^), used for power operations.

<**point**>

This non-terminal represents a decimal point (.), used in floating-point numbers.

<**relational_op**>

This non-terminal represents relational operators used for comparisons, including greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), equal to (==), and not equal to (!=).

<**type**>

This non-terminal represents data types in the program, which can be either int (integer) or float (floating-point number). Booleans are also treated as one of those, all non-zero values are treated as true, zero values are treated as false.

<**logical_op**>

This non-terminal represents logical operators, including AND (&&) and OR (||).

<**not**>

This non-terminal represents the logical NOT operator (!), used to reverse the expression.

- ## Constant Real Values

<**constant**>

This non-terminal represents a numerical constant in the program. It can be either an integer (<integer>) or a floating-point number, floating-point number consisting an integer followed by a decimal point (<point>) and an integer (<integer>).

<**integer**>

This non-terminal represents all natural numbers. The negative integers are treated as operations, when the lex file sees **-** <integer>, it will treat it as an expression and evaluate 0 - <integer>

- ## Variables

<**alphanumeric**>

This non-terminal represents a character that can either be a letter or a digit.

<**variable**>

This non-terminal represents a variable name that starts with a letter and continues with a sequence of alphanumeric characters.

<**all_char**>

This non-terminal represents every valid character in the language, including both alphanumeric values and special characters.

<**str**>

This non-terminal represents a string that can be empty or consist of one or more characters from <all_char> definiton.

<**str_with_newline**>

This non-terminal represents a string like the <str>, with an additional newline option.

- ## Expressions

<**expr**>

This non-terminal represents an expression, which is the highest-level construct that can contain logical operations, arithmetic operations, relational comparisons, function calls, and variables. Expressions are evaluated following a structured precedence.

<**or_expr**>

This non-terminal represents logical OR expressions(||). It consist of AND expressions combined using the OR operator. Logical OR has the lowest precedence among all operators, meaning it is evaluated after all other operations. It is left-associative, ensuring that expressions are grouped from left to right when multiple OR operators are present.

<**and_expr**>

This non-terminal represents logical AND expressions. It consist of relational expressions combined using the AND operator. Logical AND has a higher precedence than OR, meaning it is evaluated first when both operators are present in an expression. It is also left-associative.

<**relational_expr**>

This non-terminal represents relational expressions, which compare two arithmetic expressions using relational operators such as <, >, <=, >=, ==, and !=. Relational expressions have a higher precedence than logical AND and OR, which means that comparisons are evaluated before being used in logical operations.

<**arithmetic_expr**>

This non-terminal represents arithmetic expressions that involve addition and subtraction. These operations are evaluated after multiplication, division, and modulus operations but before relational comparisons. Arithmetic expressions are left-associative,

meaning operations are performed from left to right. They are also used for representing integers with signs.

<**term**>

This non-terminal represents terms in arithmetic expressions, which involve multiplication, division, and modulus operations. These operations have higher precedence than addition and subtraction, ensuring they are evaluated first in an arithmetic expression. Terms are left-associative.

<**factor**>

This non-terminal represents factors in arithmetic expressions. Factors include base values and exponentiation operations. Exponentiation has higher precedence than multiplication and division. It is right-associative, which means that exponentiation operations are grouped from right to left. Also the not operation can be done in here, which turns all the non-zero values 0, and all zero values into 1.

<**base**>

This non-terminal represents base values, which can be individual operands or entire expressions enclosed in parentheses. Parentheses have the highest precedence, ensuring that expressions inside them are evaluated before any other operations.

<**operand**>

This non-terminal represents the fundamental components of expressions, including variables, constants, function calls, and array elements. These are the smallest indivisible units in an expression.

- ## Declarations and Assignments

<**dec_stmt**>

This non-terminal represents the declaration statements. These includes the type and the declaration parts. Also declaration can be done with constant type.

<**dec_parts**>

Declaration parts can consist of one or more declaration parts separated by commas.

<**dec_part**>

Declaration part can be a variable, assignment statement or array declaration part.

<**array_dec_part**>

Array declaration part includes different types of array impressions. Arrays can be declared with or without initialization. When declaring arrays, their sizes may or may not be specified. The size of the array is indicated by the number written in square brackets. Also arrays can have one or more dimension.

<**arr_size_decs**>

This non-terminal represents array's sizes declaration parts. It specifies the size of the array according to the whether the array is one dimensional or multidimensional.

<**arr_size_dec**>

This non-terminal represents the array's size in brackets. The size must be natural number.

<**arr_exp**>

This non-terminal represents the expressions that are used in array initialization in curly braces.

<**arr_exps**>

This non-terminal represents the expressions that are used in two dimensional array initialization separated by commas.

<**multi_exprs**>

This non-terminal represents the expressions that are used in array initialization. It includes the all impressions that are used in one dimensional, two dimensional or multidimensional array initialization.

<**multi_expr_with_braces**>

This non-terminal represents the expressions that are used in multidimensional array initialization in curly braces.

<**multi_exprs_with_braces**>

This non-terminal represents the expressions that are used in multidimensional array initialization separated by commas.

<**assign_stmt**>

This non-terminal represents the assignment statement, it can be made with variable, the assignment operator (=), and the expression that computes the value to be assigned, like regular assignments. Also it contains array assignments and representations of an increment or a decrement of variables.

<**arr_variable**>

This non-terminal represents the specific array element. It is used when accessing an element of the array. Elements of an array can be accessed by specifying the name of the array, then the index of the element enclosed in the array subscript operator [].

<**arr_assign**>

This non-terminal represents the array assignments. It is used for assignment of a specific element of an array.

<**exprs**>

This non-terminal represents the expressions. This can consist of one or more expressions separated by commas.

<**return_stmt**>

This non-terminal represents the return statement. It consists of one of the reserved words **return** and expression.

- ## Conditional statements

<**condition_stmt**>

This non-terminal represents the conditional statements that executes different code blocks according to the conditions of the statements. Depending on whether the condition is either zero or non-zero, this structure allows the program to follow various execution pathways. It can be written 4 different forms for different uses. This can consist of one if statement, one if-else statement, if and else if statement or if, else if and else statement. These conditional statements can also contain separate conditional statements within them. The statements to be made depending on whether the condition is true or false are specified in curly brackets.

<**else_if_stmts**>

This non-terminal represents the else if statements that are used in conditional statements. This can consist of one or more else if statements.

- ## Loops

<**loop_stmt**>

This non-terminal represents all the loops within the scope of the language. The loops include two types of for loops, one “traditional” and one “for-each” loop, along with while and do-while loops. It is possible to use the for loop when the iterating integer is declared beforehand.

- **Input/output statements**

<**input_stmt**>

This non-terminal represents input operations using the **read**() function, which assigns user input to a variable.

<**output_stmt**>

This non-terminal represents output operations using the **write**() function, which prints either a string or an evaluated expression to the console.

- **Function definitions**

<**function_dec**>

This non-terminal represents the function declaration. It defines a single function with function name, function's parameters specified in parenthesis, and the body of the function (function statements) specified in curly brackets. It uses the one of the reserved words **funct** and it also can be empty. It does not have return type since we only have real numbers as return type.

<**parameters**>

This non-terminal represents the parameters that are used in the function declaration. This can be made of one parameter specified by its type and variable name or more parameters separated by commas. This can be also empty.

<**function_call**>

This non-terminal represents the function call that returns the value of the function. This consists of function name and argument list that are inputs of the function's parameters.

<**arg_list**>

This non-terminal represents the argument list that are used in the function call. They are expressions that correspond to the inputs of the function parameters. They can also be empty.

- **Comments**

<**comments**>

This non-terminal represents a sequence of comments. A program may contain a single comment or multiple comments. Comments can be used in between the statements or the functions

<**comment**>

This non-terminal represents an individual comment. A comment can either be empty (ε), a single-line comment, or a multi-line comment.

<**line_comment**>

A single-line comment starts with a # symbol, followed by a string of characters) and ends with a newline. Everything written after the # on the same line is considered a comment and is ignored.

<**multiline_comment**>

A multi-line comment starts with ##, contains one or more lines of text, and ends with another ##. It allows to write explanations containing multiple lines.

3. Terminals and Reserved Words

< : smaller than operator
> : greater than operator
<= : less than or equal to operator
>= : greater than or equal to operator
== : equals operator
!= : not equals operator
= : assignment operator
& : and operator
| : or operator
+ : addition
- : subtraction operator
* : multiplication operator
/ : division operator
% : mod operator
** : exponentiation operator
(: left parenthesis
) : right parenthesis
{ : left curly bracket
} : right curly bracket

int_main: Used for main function
funct: Use to define a function.
if: Used for if statement
else: Used for else statement
else if: Used for else if statement
for: Used for for loops
while: Used for while loops, and also at the end part of the do while loop
do: Used for do while loops
const: Used for the declaration of a constant value.
write: Used for outputs to the console.
read: Used for inputs from the console.
int: Represents integer values' type.
float: Represents floating-point values' type.

4. Evaluation of Language Design Criteria

Readability:

- Functions are defined using funct, and they accept parameters. This contributes to modularity and improves readability.
- The language allows both single-line (#) and multi-line (## ... ##) comments.
- The syntax is close to C-style languages, making it easier to read for programmers familiar with them.
- Semicolons (;) at the end of statements clearly indicate and make them easy to distinguish from each other, and increase readability.
- Curly brackets ({}) make it easy to understand the scope of blocks, especially when using conditional and loop statements.

Writability:

- The language supports both simple and compound assignment forms. For example, the presence of increment (++) and decrement (--) operators allows for more concise code.
- It includes for, while, and do-while loops, along with range-based iteration.
- The language has simple read() and write() operations.
- In conditional statements, not only logical expressions but also different expressions such as function calls, variables, arithmetic or relational expressions can be written as conditions. In this way, the grammar becomes more flexible.
- One or multidimensional arrays and the ability to access array elements with indices using square brackets make storing a data collection easy and understandable.

Reliability:

- The use of explicit types (int, float) ensures a level of type safety.
- The const keyword allows declaring constant values, reducing unintended modifications.
- The presence of if-else conditions, for loops, and while loops ensures structured and predictable execution paths.
- Since operator precedence is clearly specified in the BNF definitions and associativity rules are followed, there is no ambiguity.