VINN VINN

CS 315 Project 2 Report
Team 13

Yunus Emre Erkan 22203670 Section 1
Halil Arda Özongun 22202709 Section 1
Emir Tomrukçu 22203012 Section 1

Bilkent University

March 14, 2025

Contents

Language Name: VINN VINN

1 Complete BNF Description

<**program**> ::= <functions> kachow () <statement_block>

<**statement_block**> ::= { <statements> }

<**statements**> ::= <empty> | <statement> <statements>

<**statement**> ::= <declaration_statement> ; | <if_statement> |
<loop_statement> | <return_statement> ; | <io_statement> ; |
<assignment_statement> ; | <increment_statement> ;

<**declaration_statement**> ::= <base_declaration> | <trunk_declaration>

<**if_statement**> ::= if (<bool_expression>) <statement_block>
| if (<bool_expression>) <statement_block> else <statement_block>
| if (<bool_expression>) <statement_block> <elif_statements>
| if (<bool_expression>) <statement_block> <elif_statements> else
<statement_block>

<**elif_statements**> ::= <elif_statement> | <elif_statement>
<elif_statements>

<**elif_statement**> ::= elif (<bool_expression>) <statement_block>

<**loop_statement**> ::= <for_statement> | <while_statement>

<**for_statement**> ::= for (<init_dec_statement> ; <bool_expression> ;
<increment_statement>) <statement_block> | for (<init_dec_statement>
; <bool_expression> ; <assignment_statement>) <statement_block>

<**while_statement**> ::= while (<bool_expression>) <statement_block>

<**return_statement**> ::= return <expression>

<**io_statement**> ::= <input_statement> | <output_statement>

<**increment_statement**> ::= <identifier> ++ | <identifier> --

<**input_statement**> ::= kachin <input_id>

<**input_id**> ::= <identifier> | <identifier> , <input_id> |
<id_index_chain> | <id_index_chain> , <input_id>

<output_statement> ::= kachout <out_contents>

<out_contents> ::= <out_content> | <out_content> , <out_contents>

<out_content> ::= " <string_literal> "| <expression>

<assignment_statement> ::= <identifier> = <expression> |
<id_index_chain> = <expression>

<id_index_chain> ::= <identifier> <index_chain>

<index_chain> ::= [<expression>] | [<expression>] <index_chain>

<functions> ::= <empty> | <function> <functions>

<function> ::= <type> <identifier> (<function_parameters>) {
<statements> }

<function_parameters> ::= <function_parameter> | <function_parameters>
<function_parameter>

<function_parameter> ::= <empty> | <core_declaration> |
<core_initialization> | <trunk_parameter>

<core_declaration> ::= int <identifier> = <integer> | double
<identifier> = <double>

<core_initialization> ::= <type> <identifier>

<function_call> ::= <identifier> (<function_call_parameters>)

<function_call_parameters> ::= <empty> | <expression> | <expression>
<function_call_parameters>

<base_declaration> ::= <core_declaration> | <core_initialization> |
<base_declaration> , <additional_declaration>

<trunk_declaration> ::= trunk < <type> > <identifier> =
<trunk_initializer>
  | trunk < <type> > <identifier> = <function_call>

<trunk_initializer> ::= { <element_list> }
  | <string_literal> < <type> > (<expression>)
  | trunk < <type> > (<expression>)
  | trunk < <nested_trunk_constructor> > (<expression>)

<nested_trunk_constructor> ::= { <element_list> }
  | trunk < <type> > (<expression>)
  | <nested_trunk_constructor> > (<expression>)

<element_list> ::= <expression> | <expression> , <element_list> |
<empty>

<additional_declaration> ::= <base_type> <identifier> = <expression> |
<identifier>

<expression> ::= <arithmetic_expression> | <bool_expression> |
<function_call>

<arithmetic_expression> ::= <sum_expression>

<sum_expression> ::= <multiply_expression> | <sum_expression>
<sum_operator> <multiply_expression> | <sum_operator>
<multiply_expression>

<multiply_expression> ::= <exponent_expression> | <multiply_expression>
<multiply_operator> <exponent_expression>

<exponent_expression> ::= <base> | <base> <exponent_operator>
<exponent_expression>

<bool_expression> ::= <or_expression> | <bool_expression>
<or_expression>

<or_expression> ::= <and_expression> | <or_expression> ||
<and_expression>

<and_expression> ::= <not_expression> | <and_expression> &&
<not_expression>

<not_expression> ::= ! <not_expression> | <relational_expression>

<relational_expression> ::= <expression> <relational_operator>
<expression>| (<relational_expression>)

<relational_operator> ::= == | != | < | > | <= | >=

<identifier> ::= <letter> | <identifier> <alphanumerical>

<letters> ::= <letter> | <letter> <letters>

<alphanumerical> ::= <digit> | <letter> | _

<letter> ::= 'a' | 'b' | 'c' | ... | 'z' | 'A' | 'B' | ... | 'Z'

<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

<new_line> ::= \n

<literals> ::= <integer> | <double>

<double> ::= <integer> . | <integer> . <integer> | . <integer>

<integer> ::= <digit> | <integer> <digit>

<char> ::= <digit> | <letter> | <special_char> | ' ' | ' '

<characters> ::= <empty> | <char> | <char> <characters>

<string_literal> ::= " <characters> "

<empty> ::= ""

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

<comment> ::= <single_line_comment> | <multi_line_comment>

<single_line_comment> ::= # <characters> <new_line>

<multi_line_comment> ::= ## <multiline_content> ##

<multiline_content> ::= <empty> | <char> | <char> <multiline_content>
| <new_line> <multiline_content>

<sum_operator> ::= + | -

<multiply_operator> ::= * | / | %

<exponent_operator> ::= **

<base> ::= <literals> | <identifier> | <indexed_id> | (<expression>)
| <function_call>

<indexed_id> ::= <identifier> <index_chain>

<type> ::= <base_type> | trunk < <type> >

<base_type> ::= int | double

2 Description of Non-Terminal Literals

`<program>`
Represents the core framework of the language, defining a program that begins with the kachow function which consists of a sequence of statements.

`<statements>`
Represents a sequence of statements that are in a statement block. It can represent multiple statements recursively.

`<statement>`
Represents a single statement which can be declaration, conditional, loop, return, input/output, assignment, or special type of statement.

`<statement_block>`
Represents all of the statements of a particular class or function. It covers statements with curly braces to shape an organized statement block.

`<declaration_statement>`
Represents declaration of an integer, double, or array variables.

`<if_statement>`
Represents the structure of if and else conditional blocks. It takes a boolean condition inside parentheses and includes a statement block which will be processed if that condition is valid. There are several cases of this statement for whether there is single if, if and else, if, elif, and else statements etc.

`<elif_statements>`
Represents a sequence of elif statements. There can be multiple elif statements chained together recursively.

`<elif_statement>`
Represents the structure of if condition that is being checked if previous conditional statements are not valid. It takes a boolean condition inside parenthesis and includes a statement block which will be processed if that condition is valid.

`<loop_statement>`
Represents the types of loop statements. Loops can be either for loop or while loop.

`<for_statement>`
Represents the for loop statement. It has optional declaration or assignment of a variable to use inside the statement block. It has a conditional statement that is being

checked for processing statements inside for block. It also has a statement to advance the loop. After these parts that are inside parenthesis, there is a statement block to be processed if conditions are valid.

`<while_statement>`
Represents the while loop statement. It has a conditional statement inside paranthesis that is being checked for whether the statement block below the while statement will be processed.

`<return_statement>`
Represents the functions value returning statement. This statement can either return an expression or an identifier.

`<increment_statement>`
Represents the special increment or decrement operator. It uses "++" or "–" next to an identifier to increase or decrease its value by 1.

`<io_statement>`
Represents the types of IO statements. It can be either input or output statemenet.

`<input_statement>`
Represents the input syntax of the program. Inputs are given after "kachin" word separated by commas.

`<input_id>`
Defines the structure of input identifiers. Input identifiers can be a single identifier or multiple identifiers separated by commas, representing multiple inputs.

`<output_statement>`
Represents the output syntax of the program. Outputs are given after "kachout" statement separated by commas.

`<out_contents>`
Represents the structure of output parameters. Thanks to this, we can recursively call `<out_content>` to have empty, string, any expression or identifier.

`<out_content>`
Represents the single output content. It can be empty, string, any expression or identifier.

`<assignment_statement>`
This statement type allows us to equate identifiers with expressions or literals. In the

language, assignment statements may allow some operations and may not allow others. In assignment statements any type of real number can be equated with another type of real number. If a double number equals an integer identifier, the precission of the number is decremented, if an integer equals a double identifier, the number is saved as '.0' format. This statement does not allow assignment between trunks, but a specific number in the trunk can be used for this equalization.

`<id_index_chain>`
Gets the identifier and the index chain together to resolve the problem of accessing trunk elements outside of assignment.

`<index_chain>`
Represents the index of the element that will be accessed.

`<functions>`
Represents sequence of functions. It can be empty or contain multiple functions.

`<function>`
Represents a single function. It takes the type, identifier as well as parameters of the function which will be inside paranthesis. Statements that will be processed when the function is called are below function declaration as statements.

`<function_parameters>`
Represents a sequence of parameters that will be inside the function.

`<core_declaration>`
Represents the declarations that can be done in function parameters. It will take the type, identifier, and the value that will be assigned to the parameter intially.

`<core_initialization>`
Represents the initializations that can be done in function parameters. It will take the type and identifier of the parameter.

`<function_call>`
Represents the syntax of the function when it is called. Its identifier is given with its parameters inside paranthesis.

`<function_call_parameters>`
Represents the parameters that will be inside the function call statement. It consists sequence of identifiers or expressions that can be given as parameters.

`<base_declaration>`

Represents sequence of declarations. There can be more than one declaration recursively.

`<trunk_declaration>`
Represents the declaration of an array. It takes the type of vector inside angle brackets and trunk initializer as the right hand side of the declaration.

`<trunk_intializer>`
Represents the intialization of arrays. It can be in shape of elements of array inside brackets or trunk type and array size as an expression.

`<nested_trunk_constructor>`
Helps constructing nested multi-dimensional arrays. It completes the necessary right hand side for this syntax.

`<element_list>`
Element lists are set of literals, and these literals are used in the trunk initialization. It assigns all elements to the places in trunk. This set of real numbers can be both double or integers and they works in same way with assignment_statement while assigning elements.

`<additional_declaration>`
It makes it possible to write additional declarations or initializations of variables after the first one. Such an example would be "int x, y, z = 5". The variable "y" and "z" were initialized and declared with this rule.

`<expression>`
Represents arithmetic and boolean expressions as well as function calls. These expressions can be assigned to variables or used for operations inside the program.

`<arithmetic_expression>`
Represents a general arithmetic expression. It is divided into <sum_expression>, <multiply_expression> and <exponent_expression> to signify the precedence of some operations over the other.

`<sum_expression>`
It includes the "-" and "+" operators, which have the least precedence. For handling the matter of having negative numbers, sum_expression can have an operator and a single expression right of it. It evaluates to having a 0 on the left, which makes possible to write "-5" as a negative number.

`<multiply_expression>`

Multiply expression might have the "*", "/" and "%" operators, which have the second least precedence, or it might be only a exponent expression. It has a recursive definition to right, to ensure you have enough flexibility to make your calculations.

`<exponent_expression>`
Exponent expression is the most important expression after parantheses. It works with both integer and double numbers, for both base and power. It may call any literal, or any expression, or and expression if it is in parenthesis.

`<sum_operator>`
Represents '+' and '-' symbols for sum and subtract operations. Has the least precedence among arithmetic operations.

`<multiply_operator>`
Represents '*', '/', and '%' symbols for multiply, divide, and modulo operations. Has the least precedence among arithmetic operations.

`<exponent_operator>`
Represents '**' symbol for exponent operations. Has the most precedence among arithmetic operations.

`<bool_expression>`
Represents the expressions to use for conditionals and loops. It will return '1' when it is true and '0' otherwise. This makes it possible to act as an expression in other places such as arithmetic operations.

`<or_expression>`
Represents and boolean expressions. Its result is '1' if at least one of the expressions are true. Otherwise the result is '0'.

`<and_expression>`
Represents and boolean expressions. Its result is '1' if and only if both expressions are true. Otherwise the result is '0'.

`<not_expression>`
Represents the not operation which has the most precedence among others. It is also possible to add parenthesis. This makes them come even before the not operation which is the standard.

`<relational_operator>`
Represents relational operators to compare two expressions. It will be mainly used in conditionals and loops. It will evaluate itself to either 1 or 0.

`<identifier>`
Represents variable names that identify specific variables. It has to start with a letter and can contain underline char as well as alphanumerical characters afterward.

`<letters>`
Represents lowercase and uppercase letters from ASCII.

`<alphanumerical>`
Represents a combination of letters and digits, including underscores (‘_‘). It is commonly used for defining valid variable names and identifiers.

`<letter>`
Represents an individual alphabetical character from ’a’ to ’z’ or ’A’ to ’Z’. It forms the basis of identifiers and textual data.

`<digit>`
Represents a single numerical character from ’0’ to ’9’. It is used in numeric values and arithmetic expressions.

`<new_line>`
Represents the newline character (‘\n‘), which is used to indicate the end of a line in a text file or code.

`<literals>`
Represents numerical values in the program, which can be either integers or floating-point numbers (‘double‘ type).

`<double>`
Represents a floating-point number that contains a decimal point. It can be in the format of <integer>., <integer> . <integer>, or . <integer>.

`<integer>`
Represents a whole number without a decimal point. It can be a single digit or a sequence of digits.

`<characters>`
Represents a sequence of individual characters that can be letters, digits, special characters, whitespace, or tab.

`<char>`
Represents a single character, which can be a digit, a letter, or a special character.

`<string_literal>`
Represents the String constants that are used in kachout statements. They cannot be stored in any variable

`<empty>`
It is used to represent that a grammar can be empty.

`<comments>`
Represents a collection of comments in the program. It can be empty or consist of multiple comments, including both single-line and multi-line comments.

`<comment>`
Defines a single comment, which can be either a single-line comment or a multi-line comment.

`<single_line_comment>`
A comment that begins with the # symbol and extends to the end of the line. It is used for adding brief explanations or notes within the code.

`<multi_line_comment>`
A comment that spans multiple lines, typically starts and ends with ##. It is used for detailed explanations or documentation within the code.

`<multiline_content>`
Represents content that spans multiple lines, which can consist of characters, a combination of characters and additional multiline content, or new lines that continue the multiline sequence.

`<base>`
It represenents identifiers, expressions and literals. This base can be used in any of the arithmetic expressions and can be passed to function calls as a parameter.

`<indexed_id>`
It represenents identifier with an index chain.

`<type>`
Defines the type of a variable or function. It can be a basic type (int, double) or a structured type such as trunk<type>, indicating a more complex or user-defined type.

`<base_type>`
Represents the fundamental data types available in the language. The base types include ‘int‘ for integers and ‘double‘ for floating-point numbers.

3 Terminals

< : 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 and string concatenation operator
- : Subtraction operator
* : Multiplication operator
/ : Division operator
% : Mod operator
** : Exponentiation operator
++ : Increment operator
– : Decrement operator
(: Left parenthesis
) : Right parenthesis
{ : Left curly bracket
} : Right curly bracket
[: Left square bracket
] : Right square bracket
; : Semicolon
, : Comma
: Comment symbol
: Multiline comment symbol
" : Quotation mark

4 Nontrivial Tokens

`kachow:`
The entry point of the program where execution begins. It serves as the starting function or main block from which all other code flows.

`kachin:`
Used to take input from terminal. It acts as a keyword that signals next tokens will accept input and store the input value.

`kachout:`
Used to give output to terminal. The tokens after it will be displayed in the terminal.

`function:`
Defines a reusable block of code that performs a specific task. Functions can take inputs, process them, and return outputs.

`if:`
Used to introduce a conditional statement. It executes a block of code only if a specified condition evaluates to true.

`elif:`
Short for "else if," it allows checking additional conditions if the previous 'if' condition was false.

`else:`
Defines an alternative block of code that runs when none of the preceding 'if' or 'elif' conditions are met.

`for:`
A loop construct that iterates over a sequence, such as a list or a range of values, executing its block repeatedly.

`while:`
Executes a block of code repeatedly as long as a specified condition remains true.

`int:`
Represents an integer data type, which stores whole numbers without decimal points.

`double:`
A floating-point data type that provides more precision than an 'int', allowing storage of decimal values.

`return:`
Used inside a function to terminate execution and optionally send a value back to the caller.

`trunk:`
A dynamic array that can change in size during program execution, allowing elements to be added or removed.

5 Evaluation of Language Design Criteria

5.1 Writability

Our language is very similar in syntax to other c type languages. This makes it intuitive for people coming from other languages. In addition, it supports operations such as incerement(++) and decrement(--) to reduce complexity. It also allows both direct variable initialization and separate declarations, which makes it easier to write. We also have an array-like data structure called trunk. With this data structure other data structures can be defined. On the other hand, some of the keywords in our language (kachow, kachin, kachout, etc.) can be difficult to write for people from other languages. Also, the fact that we have an array-like data structure and no other data structure makes it difficult to use the language. In addition, since lambda expressions are not available in our language, it is necessary to constantly open blocks.

5.2 Readability

The input output keywords, and data structure keyword of the language makes the language difficult to read at first, but the "in" and "out" phrasings in these expressions make these expressions intuitive. Also, expressions like if, elif, else explain how conditional expressions work well, and the inability to use lambda expressions prevents these expressions from being difficult to read. The expressions used for loops explain how loops work very well, but the part of the for loop inside the parentheses may not be very readable for those seeing it for the first time. Arithmetic, boolean, and assignment expressions follow standard precedence, which improves readability. It also uses traditional arithmetic and boolean operators (+, -, *, /, &&, ||), making the language familiar to the user by using familiar expressions. Finally, the lack of a bool data structure and the fact that bools are stored as int makes the language difficult to read for the user.

5.3 Reliability

The types of variables created in the language are specified, by this type safety it ensures that any errors are found at compile time, reducing the likelihood of runtime problems and increasing the overall reliability of the language. The use of curly braces for scopes in the language helps early detection of errors in the language for scopes. A semicolon after statements also helps with this. On the other hand, the requirement to specify the size of data structures when creating them makes the language difficult to use, but it also increases its reliability. Also, the well-defined grammar of the language leaves no room for ambiguity and the expressions written in the language are understood by everyone in the same and unified way. A factor that reduces reliability in a language is the lack of error handling mechanisms. Lack of such expressions could lead to runtime crashes without proper debugging tools.