
# Bilkent University
# CS 315
# Project 2 Report

Language: Siu

Team 8
Sarp Yıldırım 22003330 Section 1
Göktuğ Yılmaz 21903048 Section 1
Mehmet Emre Güneş 22003287 Section 1

# Part A - Language Design

## 1. Initial Program

<siu> ::= @BEGIN <stmt_list> @END

<stmt_list> ::= <stmt> | <stmt> <stmt_list>

<stmt> ::= <if_stmt> | <declaration> | <assignment> | <declaration_assignment> | <loop_stmt> | <func> | <call_func> <input_stmt> | <output_stmt> | <comment>

<if_stmt> ::= if <LP> <expression> <RP> <LCB> <stmt_list> <RCB> | if <LP> <expression> <RP> <LCB> <stmt_list> <RCB> else <LCB> <stmt_list> <RCB>

## 2. Variables

<array> ::= <LCB> <elements> <RCB>

<elements> ::= <element> | <element> <comma> <elements>

<element> ::= <bool> | <var>

<type_name> ::= bool | array

<any_type> ::= <bool> | <var> | <array>

<bool> ::= True | False

<var> ::= <string>

## 3. Assignment and Declarations

<assignment> ::= <var> <assign_op> <expression> | <var> <assign_op> <array>

<declaration_assignment> ::= <declaration> <assign_op> <expression>

<declaration> ::= <type_name> <var>

## 4. Logical Conditions

<expression> ::= <cond_expr_or>

<cond_expr_or> ::= <cond_expr_and> | <cond_expr_and> <or_op> <cond_expr_or>

<cond_expr_and> ::= <cond_expr_not> | <cond_expr_not> <and_op> <cond_expr_and>

<cond_expr_not> ::= <cond_expr> |<not_op> <cond_expr_not>

<cond_expr> ::= <cond_expr2> | <cond_expr2> <relate_op> <cond_expr2>

<cond_expr2> ::= <var> | <bool > | <call_func> | <LP> <expression> <RP>

<relate_op> ::= <equal_op> | <not_equal_op> | <implication_op> | <double_implication_op>

## 5. Loops

<loop_stmt> ::= <for_loop> | <while_loop>

<for_loop> ::= for <LP> <type_name> <var> in <var> <RP> <LCB> <stmt_list> <RCB>

<while_loop> ::= while <LP> <expression> <RP> <LCB> <stmt_list> <RCB>

## 6. Functions

<func> ::= <non_void_func> | <void_func>

<non_void_func> ::= <type_name> <var> <LP> <parameters> <RP> <LCB> <stmt_list> return <return_stmt> <RCB>

<void_func> ::= void <var> <LP> <parameters> <RP> <LCB> <stmt_list> <RCB>

<parameters> ::= <empty> | <declaration> | <declaration> <comma> <parameters>

<return_stmt> ::= <expression>

<call_func>::= <var> <LP> <RP> | <var> <LP> <arguments> <RP>

<arguments> ::= <var> | <var> <comma> <arguments> | <any_type> | <any_type> <comma> <arguments>

## 7. Input and Output

<input_stmt> ::= insiu <LP> <var> <RP> <LCB> <QM> <sentence> <QM> <RCB>

<output_stmt>::= siu <LCB> <output_context> <RCB>

<output_context> ::= <QM> <sentence> <QM> | <var>

## 8. String Constants

<string> ::= <char_list> | <char_list> <digit_list> | <char_list> <digit_list> <string>

<char_list> ::= <char> | <char> <char_list>

<digit_list> ::= <digit> | <digit> <digit_list>

<char>::= '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'

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

<char_all> ::= <char_list> | <char_special> | <digit_list> | <char_list> <char_all> | <char_special> <char_all> | <digit_list> <char_all>

<char_special> ::= ! | @ | # | \$ | % | ^ | & | * | ( | ) | + | = | / | * | - | ' | " | ; | '|' | { | } | [ | ] | .| ,

<sentence> ::= <char_all> | <space> | <char_all> <sentence> | <space> <sentence>

## 9. Comments

<comment> ::= /* <sentence> */

## 10. Symbols

<assign_op> ::= =

<equal_op> ::= ==

<not_equal_op> ::= !=

<and_op> ::= &

<or_op> ::= |

<not_op> ::= !

<implication_op> ::= ->

<double_implication_op> ::= =>

<LB> ::= [

<RB> ::= ]

<LP> ::= (

<RP> ::= )

<LCB>::= {

<RCB>::= }

<comma> ::= ,

<QM> ::= "

<new_line> ::= \n

<empty>::= ""

<space> ::= " "

## Explanations

**<siu> ::= @BEGIN <stmt_list> @END**

This is the start symbol. It is the initial state to start the program. The execution will be done inside the @BEGIN and @END.

**<stmt_list> ::= <stmt> | <stmt> <stmt_list>**

This non-terminal is for statements which does not have any amount limit.

**<stmt> ::= <if_stmt> | <declaration> | <assignment> | <declaration_assignment> | <loop_stmt> | <func> | <call_func> <input_stmt> | <output_stmt> | <comment>**

This non-terminal states that a statement can be an if statement, declaration, assignment, both declaration and assignment, a loop, a function, a function call, an input, an output or a comment statement.

**<if_stmt> ::= if <LP> <expression> <RP> <LCB> <stmt_list> <RCB> | if <LP> <expression> <RP> <LCB> <stmt_list> <RCB> else <LCB> <stmt_list> <RCB>**

This non-terminal is indicating how to write an if statement. It starts with if keyword and between parentheses the condition expression is written. After that, between the curly braces the statement list that will be executed is written. Optionally, else block can be added which may be done by writing else keyword and between curly braces writing the statement list that will be executed if the condition in if block is not true.

**<any_type> ::= <bool> | <var> | <array>**

This non-terminal is showing every type of data in the program.

**<array> ::= <LCB> <elements> <RCB>**

This non-terminal is for writing arrays. An array needs elements between curly braces.

**<elements> ::= <element> | <element> <comma> <elements>**

This non-terminal is for the elements for the array. It can be single or multiple, if there are more than one element than all of them need to be seperated with commas.

**<element> ::= <bool> | <var>**

This non-terminal is for element stating that it can be either a variable or a boolean.

**<type_name> ::= bool | array**

This non-terminal states that type of variable can be either a boolean or an array which also be used as return type of functions.

**<bool> ::= True | False**

This non-terminal states that a boolean can be either true or false

**<var> ::= <string>**

This non-terminal states that a variable name is a string.

**<assignment> ::= <var> <assign_op> <expression> | <var> <assign_op> <array>**

This non-terminal states that an assignment statement can written with variable name followed by assignment operator and an expression or an array.

**<declaration_assignment> ::= <declaration> <assign_op> <expression>**

This non-terminal states that both declaration and assignment statement can be written as a declaration followed by an assignment operator and followed by an expression.

**<declaration> ::= <type_name> <var>**

This non-terminal states that a declaration can be written as type of the variable followed by the name of variable.

**<expression> ::= <cond_expr_or>**

This non-terminal states that an expression derives as a conditional expression with or operator.

**<cond_expr_or> ::= <cond_expr_and> | <cond_expr_and> <or_op> <cond_expr_or>**

This non-terminal states that conditional expression with or operator can derive as a conditional expression with and operator or can derive into one and expression and one or expression combined with or operation

**<cond_expr_and> ::= <cond_expr_not> | <cond_expr_not> <and_op> <cond_expr_and>**

This non-terminal states that conditional expression with and operator can derive as a conditional expression with not operator or can derive into one not expression and one and expression combined with and operation

**<cond_expr_not> ::= <cond_expr> |<not_op> <cond_expr_not>**

This non-terminal states that conditional expression with not operator can derive as a conditional expression or can derive into not operation followed by a not expression

**<cond_expr> ::= <cond_expr2> | <cond_expr2> <relate_op> <cond_expr2>**

This non-terminal states that conditional expression may be another conditional expression or two conditional expressions that are combined with relation operations.

**<cond_expr2> ::= <var> | <bool> | <call_func> | <LP> <expression> <RP>**

This non-terminal states that conditional expression can be a variable, a boolean or a return element of a function or result of any combination of logical expressions.

**<relate_op> ::= <equal_op> | <not_equal_op> | <implication_op> | <double_implication_op>**

This non-terminal states that a relation operation may be equals operation or not equals operation, implication or double implication operations.

**<loop_stmt> ::= <for_loop> | <while_loop>**

This non-terminal states that a loop statement can be either a for loop or a while loop.

**<for_loop> ::= for <LP> <type_name> <var> in <var> <RP> <LCB> <stmt_list> <RCB>**

This non-terminal states to write a for loop, first for keyword is needed followed by parantheses. In between parentheses, data type of the variable is needed followed by a variable name which is followed by keyword in and lastly the name of the variable which will be iterated. After this, between the curly braces, the statment list can be written.

**<while_loop> ::= while <LP> <expression> <RP> <LCB> <stmt_list> <RCB>**

This non-terminal states that a while loop can written starting with keyword while followed by parentheses. Between the parentheses, logical condition is writtend and between the curly brackets the statement list is written.

**<func> ::= <non_void_func> | <void_func>**

This non-terminal states that a function can be a non-void function or a void function.

**<non_void_func> ::= <type_name> <var> <LP> <parameters> <RP> <LCB> <stmt_list> return <return_stmt> <RCB>**

This non-terminal states that a non-void function can be written as following: First type name of the returned element is needed followed by the name of the function. After that, between the parentheses the parameters of the function is written followed by curly braces. Between curly braces the statement list is written and lastly return keyword is used for returning an element followed by the element which will be returned.
**<void_func> ::= void <var> <LP> <parameters> <RP> <LCB> <stmt_list> <RCB>**

This non-terminal states a void function can be written starting with void keyword followed by the name of the function. After that, between the

parentheses the parameters of the function is written followed by curly brackets in which between of them the statement list is written.

**<parameters> ::= <empty> | <declaration> | <declaration> <comma> <parameters>**

This non-terminal states that the parameters of a function can be empty, any declaration. If declarations are more than one, they need to be seperated by commas.

**<return_stmt> ::= <expression>**

This non-terminal states that any expression can be returned by a function.

**<call_func>::= <var> <LP> <RP> | <var> <LP> <arguments> <RP>**

This non-terminal shows how to call a function. First, name of the function is writted followed by parentheses and if that function has any arguments, they are written between the parentheses.

**<arguments> ::= <var> | <var> <comma> <arguments> | <any_type> | <any_type> <comma> <arguments>**

This non-terminal states that arguments of a function may be a single variable or multiple variables in which they need to be seperated by commas. This also applies to any type of data.

**<input_stmt> ::= insiu <LP> <var> <RP> <LCB> <QM> <sentence> <QM> <RCB>**

This non-terminal states that an input from user can be taken if insiu keyword is written followed by parentheses and between the parentheses the variable is needed. After that, Between curly braces and quotation marks the sentence can be written that will bi displayed to user while asking input.

**<output_stmt>::= siu <LCB> <output_context> <RCB>**
This non-terminal states that any message can displayed to the user by keyword siu followed by curly braces and between the curly braces the output context is written.

**<output_context> ::= <QM> <sentence> <QM> | <var>**

This non-terminal states that output context can be either a variable whose value will be displayed or any sentence that is between quotation marks.

**<string> ::= <char_list> | <char_list> <digit_list> | <char_list> <digit_list> <string>**

This non-terminal states that a string may consist of any combination of characters and digits except that it cannot start with a digit.

**<char_list> ::= <char> | <char> <char_list>**

This non-terminal states that a character list is either a single character or any combination of characters.

**<digit_list> ::= <digit> | <digit> <digit_list>**

This non-terminal states that a digit list is either a single digit or any combination of digits

**<char>::=**
**'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'**

This non-terminal states what a character may be.

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

This non-terminal states what a digit may be.

**<char_all> ::= <char_list> | <char_special> | <digit_list> | <char_list> <char_all> | <char_special> <char_all> | <digit_list> <char_all>**

This non-terminal states that char_all is any combination in which all characters are available.
**<char_special> ::= ! | @ | # | \$ | % | ^ | & | * | ( | ) | + | = | / | * | - | ' | " | ; | '|' | { | } | [ | ] | .| ,**

This non-terminal states the special characters that are not letters.

**<sentence> ::= <char_all> | <space> | <char_all> <sentence> | <space> <sentence>**

This non-terminal states that a sentence may be any combination of any character including spaces.

**<comment> ::= /* <sentence> */**

This non-terminal states that a comment can be written as a sentence that is enclosed with reserved symbols.

**<assign_op> ::= =**

This non-terminal is for assignment operator. The operator is an equals  sign. This indicates that the variable in the left hand side of the operator has the data of the right hand side.

**<equal_op> ::= ==**

This non-terminal is for comparing two variables whether they are equal or not. The operator is two equals signs and it checks whether the left hand side's data is the same as the right hand side's data.

**<not_equal_op> ::= !=**

This non-terminal is for comparing two variables whether they are equal or not. The operator is combination of one not operator followed by an equal sign and it checks whether the left hand side's data is differs from the right hand side's data.

**<and_op> ::= &**

This non-terminal is for and operation. It is an ampersand symbol. It checks whether both the right hand side and the left hand side's values are true.

**<or_op> ::= |**

This non-terminal is for or operation. The symbol is a bar. It checks if either of left hand side or right hand side's value is true.

**<not_op> ::= !**

This non-terminal is for not operation. The sign is an exclamation mark. It inverses the data on the right hand side of itself.

**<implication_op> ::= ->**

This non-terminal is for implication operation. This gives a true result if the left hand side implies the right hand side of the operator. If not, it gives false as a result.

**<double_implication_op> ::= =>**

This non-terminal is for double implication operation. This gives a true result if the left hand side doubly implies the right hand side of the operator. If not, it gives false as a result.

**<LB> ::= [**

This non-terminal is simply for a left bracket.

**<RB> ::= ]**

This non-terminal is simply for a right bracket.

**<LP> ::= (**

This non-terminal is simply for a left parenthesis.

**<RP> ::= )**

This non-terminal is simply for a right parenthesis.

**<LCB>::= {**

This non-terminal is simply for a left curly bracket.

**<RCB>::= }**

This non-terminal is simply for a right curly bracket.

**<comma> ::= ,**

This non-terminal is simply for a comma.

**<QM> ::= "**

This non-terminal is for quotation marks that are used for string constants.

**<new_line> ::= \n**

This non-terminal is for a new line.

**<empty>::= ""**

This non-terminal is for emptiness.

**<space> ::= " "**

This non-terminal is for a space.

## Explanation of Non-Trivial Tokens

**@BEGIN:** This non-trivial token is used to show the beginning of the execution.

**@END:** This non-trivial token is used to show the end of the execution.

**void:** This non-trivial token is used for return type of functions as void.

**if:** This non-trivial token is used for if statements.

**else:** This non-trivial token is used for if-else statements.

**while:** This non-trivial token is used for while loops.

**for:** This non-trivial token is used for for each loops.

**siu:** This non-trivial token is used to give an output to the user.

**insiu:** This non-trivial token is used to take an input from the user.

**return:** This non-trivial token is used for returning from a function.

**in:** This non-trivial token is used in for each loops to specify which variable, to iterate in.

**true:** This non-trivial token is used for booleans' data which are true.

**false:** This non-trivial token is used for booleans' data which are false.

**bool:** This non-trivial token is used to declare a boolean variable.

**array:** This non-trivial token is used to declare an array.

**/* ... */:** This non-trivial token is used for comments which is only a single line comment and in that line only the comment should be.

## Execution of the Program:

The program's execution is determined with two non-trivial tokens which are @BEGIN and @END. The code between these two tokens are executed. If it matches the code as a function definition, it will keep its line number until @END is reached. For any function calls, these line numbers are used. Otherwise, it will keep going line by line until @END is reached.

## Handling Input and Output:

In this program, it is possible to take input from the user via reserved word insiu and display a message to the console via reserved word siu. Since this program only operates on boolean and array data types, if the user submits any data type other than boolean such as string, integer, or float, the program will crash and must be rerunned. For displaying messages, it is only possible to use string for messages and boolean data type or variables for their values. If a user tries to display a message containing a float or integer that is not in the type of string, the program will crash and need to be rerunned.

## Conventions:

While writing blocks, there is no need for indentation but lowercase, and uppercase should match for loops, variables, if statements, etc. Brackets, curly brackets, and parentheses are essential to write input/output statements, if and else statements, functions, loops, and

array declarations. All of the functions use pass-by-value. Due to this, if any change is needed to be done to a variable outside the function, it should be done with the return values of functions. Otherwise, the changes in the function will only change the local copy of the arguments. Moreover, due to the line by line execution of the program functions need to be defined above the code that will call the function so that program will know the functions line number when its called. If function is defined under the caller, program will not be able to identify the function.

## Evaluation of Siu Programming Language

## Readability:

Siu has limited statements and so reading is quite easy especially for someone who knows a C-like language since curly brackets are used commonly in the language. Moreover, only having one syntax for a specific statement and not having any alternatives makes it easier to understand.

## Writability:

Siu has a C-like program syntax with curly brackets, functions, while loop, if statements and has a for each loop which is simpler than C. Other statements like assigning and declaring variables are also are common types of notion that are seen most of the languages. As a result, it is easy to write if you are familiar with Java or C and otherwise, since it has few statements, it is easy to learn.

## Reliability:

Since Siu has a limited variables and statements it is both easy to write and read which makes debugging and fixing errors and bugs easier.