CS 315 - Programming Languages
<u>Project 1</u>

Group Members TEAM 10:
Ghulam Ahmed <22101001>
Mehshid Atiq <2211335>
Muhammad Rowaha <22101023>
Computer Engineering

Date: 14.10.2023

RGMScript

Extended Backus-Nour Form

The following EBNF defines the syntax of our language. The top-level construct–the start symbol–is <program>, which is defined in terms of either function definitions or a list of statements. A similar construct can be found in the Python programming language. However, our language uses the **begin** and **end** keywords for scopes (similar to Systemverilog) since we believe that since keywords provide a more detailed and transparent view of the flow of the code. Yet, to enforce good coding practices, our statements will always end in a newline character–that is to say that a statement will only be evaluated once a newline is reached.
Below is the definition of the program rule:

```
<program> ::= {<function_definition> | <statement_list>}
```

To define the syntax further, defining the fundamental rules is paramount. Since the language contains only the integer datatypes, below is the rule defining the syntax of integer literals and variables. An integer is a sequence of numbers and can be signed or unsigned. A variable is a sequence of letters, digits and underscore but should always either start from underscore or a letter. We make a distinction between string and prompt in our language. Although explained further in detail, this allows us to meet the requirements of the language by calling a sequence of any printable character a string, a string enclosed in double quotes a prompt for I/O operations, and a string enclosed in a pair of forward slash and asterisk (/* and */) a comment.

```
<variable> ::= (_ | <letter>) {<letter> | <digit>}
<integer> ::= [<sign>] <digit> {<digit>}

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 7 | 8 | 9
<sign> ::= + | -
<letter> ::= <uppercase> | <lowercase>
<uppercase> ::= 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
<lowercase> ::= 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

<prompt> ::= “<string>”
```

```
<string> ::= <character> | <string> <character>
<character> ::= <any printable character except ">
```

For example, 12, -12 and +12 are all valid integer literals. And, _12, a12, _a12 are all valid variables (variable identifiers). Providing a rule to define of list of variables and/or integer literals is useful in defining more abstract rules such as function parameters and integer arrays. Although these rules do not themselves have a token or do not act as lexemes, they do allow us to define more complex rules like array elements and function parameters.

```
<combined_list> ::= <variable_list> | <integer_list> | <combined_list> ,
(<variable_list> | <integer_list>)
<variable_list> ::= <variable> | <variable_list> , <variable>
<integer_list> ::= <integer> | <integer_list> , <integer>
```

For instance, *12, 13, 14* is a valid integer list, and *_12, _a12, a12* is a valid variable list. Therefore *12, 13, 14, a12, _a12, _12* is a valid combined list.

The first use-case of this rule can now be used to define integer arrays and their built-in operations in our language. The integer array construct has a combined list in a pair of square brackets and will be a lexeme in our language (with an associated token ARRAY). It will have associated methods and those method calls will also be lexemes (with an associated token of ARRAY_METHOD). We also support an index operator in our language and its syntax closely relates to contemporary programming languages.

```
<int_arr> ::= [<combined_list>]
<int_arr_method> ::= <variable>.<variable>([combined_list])
<int_arr_idx> ::= <variable>[(<variable>|<integer>)]
```

[12, 13, 14, a12, _a12, _12] is a valid integer array. Since the integer array is the only structured type in the language, it will have associated methods with it; their syntax is defined in the <int_arr_method> rule. Examples can be the following: the append operation *(called with **myarray.append(2)**)* will allow appending an element to the end of the list. The remove operation will allow to remove an element by index, the size operation will return the size of the array, and the at operation will allow to get an element by index. Note that the index can be negative; in that case, the element will be determined by size() - index.

Using these built-in types (integers and integer arrays), the assignment statement in the language can be defined as follows:

```
<assign_statement> ::=  <expr_assign> | <array_assign> | <input_assign>
<array_assign> ::= <variable> := <int_array>
<expr_assign> ::= <variable> := <expresssion>
<input_assign> ::= <variable> := input <prompt>
<expression> ::= <term> | <expression> (+ | -) <term>
<term> ::= <base> | <term> (* | / | %) <base>
<base> ::= <exponent> | <base > ^ <exponent>
<exponent> ::= <variable> | <integer> |  <function_call> | <int_arr_method> |
<int_arr_idx> | (<expression>)
```

A variable can be initialized as follows:

abc := 12
abc := [12, 13, 14, 15]
abc := input "enter value of abc: "

The assignment operator := has the lowest precedence followed by + (addition) , - (subtraction). Operators * (multiplication), / (integer division), and % (modula) have higher precedence and the operator ^ (exponentiation) has the highest precedence. However, using a matching pair of round parenthesis, any operator's precedence can be increased (except the assignment operator), and the expressions in the brackets will always be evaluated first. The basic unit of an expression is an exponent which can either be a variable, integer, integer array methods, integer array index, a nested expression in a pair of round parenthesis, or a function call. This also defines an in-built operator **input** that can prompt a user and return an integer input by the user. The **input** function will block the expression evaluation until the user inputs. The call *input "enter value for a: "* is a valid input procedure call and will return a value input by the user.

Below are the rules for function definitions and function calls. Note that function definitions define the top-level program rule.

```
<function_call> ::= <variable>([combined_list])
<function_definition> ::= <variable> ([<variable_list>]) : <variable>
<function_block>
<function_block> ::= begin <statement_list> return <variable> <end>
```

Below is an example of a valid function definition:

```
add(a, b, c) : d begin
    d := a + b  + c
    return d
end
```

And this function can be called in either of the following ways: *add(a, b, c), add(1, 2, 3)* or *add(a, 1, 2)*. Like the assignment statements, the parameters also do not have any keyword to denote their data type since that would be redundant as the language only supports the integer data type (the distinction between an integer and an integer array is made distinct by the way they are declared). Our function signatures mimic the Golang functions quite closely. However our functions cannot return multiple return values, the variable after the colon can be used inside the function block and will be implicitly initialized during any function calls.

Next, looping and conditional rules are defined. A basic comparison operation can be either a signed or an unsigned comparison. This allows the user to compare two integers by their magnitudes only.

```
<comparison_expression> ::= <expression> <comparison_operator> <expression>
<comparison_operator> ::= <signed_comparison> | <unsigned_comparison>
<unsigned_comparison> ::= $<signed_comparison>
<signed_comparison> ::= == | < | > | != | >= | <=
```

Then, the *if*, *if else*, *while,* and *for* can be defined as follows:

```
<if_statement> ::= if (comparison_expression) begin <statement_list> end |
                   if (comparison_expression) begin <statement_list> end else
begin <statement_list> end

<while_statement> ::=  while (comparison_expression) begin <statement_list> end
|
                  while (comparison_expression) begin <statement_list> end else
begin <statement_list> end // else only runs if while was initially false

<for_statement> ::= for (<expr_assign>, <comparison_expression>,
<expr_assign>) begin <statement_list> end
```

Below are examples for *if* and *if else*.

```
if (a + b + c %> d -12 + foo()) begin
        f := 14
end else begin
        f := -14
end
```

The while loop also has an else block, but unlike Python, the else block will only be evaluated once if the while loop was initially false. Consider the following example use-case (this uses the print built-in function to print to the terminal which is defined later):

```
a := input “enter a value for a: ”
while (a > 12) begin
        a := a - 1
end else begin
        print(“a was greater than 12“)
end
```

Since now all the possible statements have been defined, they can be combined to define a statement rule and then a statement list itself (a collection of statements). Notice that the statement list is used to further define the <program> start symbol, which, hence, completes the definition of the language.

```
<print_statement> ::= display {(<prompt>|<expression>)}
<comment_statement> ::= \* <string> *\

<statement> ::= <assign_statement> | <if_statement> | <while_statement> |
<for_statement> | <print_statement> | <int_arr_method> | <comment_statement>
<statement_list> ::= <statement> | <statement_list> <statement>
```

TOKENS and LEXICAL ANALYZER

The following is the list of tokens in the language is as follows:

LEXEME	TOKEN
input	INPUT_OP
display	OUTPUT_OP
begin	SCOPE_OPEN
end	SCOPE_CLOSE
if	IF
else	ELSE
while	WHILE

return	RETURN
for	FOR
:=	ASSIGN_OP
+	ADD_OP
-	SUBTRACT_OP
*	MULTI_OP
/	DIV_OP
%	MODULA_OP
^	EXPONENT_OP
(	BRACKET_OPEN
)	BRACKET_CLOSE
==	EQUAL_OP
!=	UNEQUAL_OP
>	GREATER_OP
<	LESSER_OP
>=	GREATER_EQUAL_OP
<=	LESSER_EQUAL_OP
$==	US_EQUAL_OP
$!=	US_UNEQUAL_OP
$>	US_GREATER_OP
$<	US_LESSER_OP
$>=	US_GREATER_EQUAL_OP
$<=	US_LESSER_EQUAL_OP
,	COMMA_OP
\n	STMT_END
<integer>	CONST
<variable>	VARIABLE

<int_array>	ARRAY
<prompt>	STRING
<comment_statement>	COMMENT
<function_call>	CALL_OP
<function_definition>	FUNC_SIG
<int_arr_method>	ARRAY_METHOD
<int_arr_idx>	INDEX_OP
EOF	PROGRAM_END