Team members:

Ragiri Basava Rudreshwar (21110172)
Keerthi Ramineni (21110176)
Sudharshan Kumar Bhardwaj (21110218)

ZEVA

Syntax of the language:

1. **Basic Types (numbers, booleans, strings):** Declaration format: var: <type>

var: <type> x= <data>; @ we have to always end a statement with a semicolon(;)
<type> :=int @integer (1,2)
|str @string ("compiler")
|bool @boolean (true,false)
|float @floating point numbers
|double @decimal numbers with more digits
|long @big number with exceed the int data type

var: str s;
s.substring(start,length) @we have to give the starting index and the length of the sub string we require
var: str c;
s.con(c); @for concatenating two strings we have to give as string_1.con(string2)

zout (x); @prints x;
<unary-operator> ::= ++ @increment by 1
| — @ decrement by 1
| & @address of an operator
| - @unary minus operator
| ~ @compliment
| ! @logical not operator

<binary-operator> ::= + @addition
| - @subtraction
| * @multiplication
| / @division
| % @remainder
|== @comparision of identifier with the required value
|< @less than symbol
|> @greater than symbol
|<= @less than equal symbol

|>= @greater than symbol

@ single line comment
!!! multiple comments !!!
Identifiers: alphanumeric

2. **Compound Types** :
tuple <Identifier>=(data); @they are immutable
array <Identifier>< <type> > = {data}; @same data type elements are only stored
list <Identifier>=[]; @ all the three have 0 based indexing
!!!
For both tuple and array same basic operations are applied
<Identifier>.add(data); @adds data to the compound type
<Identifier>.delete(); @removes the last data of the compound type
size=<Identifier>.size();
var: <type>a=<identifier>[1]; @ for accessing the second element in the array or tuple here type is the type
<Identifier>.front(); @ gives the first element of the compound type
<Identifier>.rear(); @giving the last element of the compound type
!!!

3. **Conditionals :**

```
if (@condition)
begin
    @code
end
elif(@condition)
begin
    @code
end
else
begin
    @code
end
```

@nested if statements

```
if(@condition)
begin
    @code
     if(@condition2)
     begin
          @code
      end
    elif(@condition)
begin
```

```
        @code
    end
    else
    begin
        @code
    end

end
```

4. **Loops :**

```
   while(condition)
  begin
        @code
   end
@ we can also use nested loops
    while(condition)
    begin
        @code
         while(condition)
         begin
              @code
         end

  end
```

5. **Branching statements :**

```
    break; @breaks the loops
    continue; @ continue to the next step of the loop
    return; @ returns the element given there
   Example :
    Var: int i=0;
     var : int n=5;
    while(i<n)
    begin
       i++;
       if(i==3)begin break  end;
    end            @here it breaks the loop when i=3
```

6. **Functions :**

```
  <type>|tuple|list|array myfunction(arguments) @example myfunction(int x,int y)
  begin
       @function body
   return data;
  end
  myfunction(x,y); @ for calling function again anywhere after the function declaration
```

7. **Closures :**

```
<type>|tuple|list|array myfunction(int x,int y)
begin
    var: x=5;
    var: y= 6;
    <type>|tuple|list|array myFunction(int a)
    begin
        var: a=7;
        @function body
        var: int output=x=y+a;
        return output;
    end
    return myFunction;
end
```

8. **Mutable variables :**

```
var: <type> | lists @ these are mutable variables
```

9. **Exceptions :**

```
try
 begin
   @ try the case
   @if fails throw the exception
 end
except(exception)
 begin
    @do the code given here
 end
```

<program> ::= <statement>*

<statement> ::= <variable_declaration>| <assignment> | <conditional> | <loop> | <function> | <try-catch>|<print_statement>|<mutable_variable_declaration>

<variable_declaration> ::= {‘var:’ <type> <identifier> ‘=’ <expression> ‘;’} + {‘tuple’ <identifier> ‘=’ <expression> ‘;’} + {‘array’ <identifier> ‘=’ <expression> ‘;’} + {‘tuple’ <identifier> ‘=’ <expression> ‘;’}

<assignment> ::= <identifier> "=" <expression> ‘;’

<conditional> ::= {{"if" <expression> "begin" <statement> “end”}* "else" “begin” <statement> “end”} + {"if" <expression> "begin" <statement> “end” {"elif" “begin” <statement> “end”}* "else" “begin” <statement> “end”}

<loop> ::= "while" <expression> “begin” <statement> “end”

<function> ::= <type>|tuple|list|array|void <identifier> "(" <identifier-list>? ")" "begin" <statement>* “return” <expression> “;” "end" | <function>

<try-except> ::= "try" "begin" <statement>* "end" "except" "(" <identifier> ")" "begin" <statement>* "end"

<print_statement> ::= 'zout' '(' expression ');'

<mutable_variable_declaration> ::= “var:” <type>| “list” <identifier> '=' <expression> ';'

<expression> ::= <number> | <boolean> | <string> |unary_operation| <identifier> | "(" <expression> ")" |< function_call> | <list_operation> | <array_operation> |< member_access>|<term> { <binary_operator> <term> }*

<unary-operator> ::= ++
| —
| &
| - r
| ~
| !

<binary-operator> ::= +
| -
| *
| /
| %
|==
|<
|>
|<=
|>=

<term> ::= <factor> { <binary_operator> <factor> }*

<factor> ::= <number> | <boolean> | <string> | <identifier> | "(" <expression> ")"

<identifier-list> ::= <identifier> { "," <identifier> }*

<list_operation> ::=< identifier> '.' ('add' '(' expression ')' | 'size' '(' ')' | '[' expression ‘]’ | 'head' '(' ')' | 'tail' '(' ')')

<array_operation> ::= <identifier> '.' ('add' '(' expression ')' | 'size' '(' ')' |'[' expression ‘]’ | 'head' '(' ')' | 'tail' '(' ')')

<member_access> ::= <identifier> '.' <identifier>

<identifier> ::= <letter> { <letter> | <digit> }*

<number> ::= <digit>+

<boolean> ::= "true" | "false"

<string> ::= "" { <character> }* ""

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

<digit> ::= "0" | "1" | "2" | ... | "9"

<character> ::= <letter> | <digit> | <special-character>

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