Language Manual: miniC

1. Data Types

➢ This language supports the following primitive data types:
• integers • character • boolean

➢ The aggregation of the aforementioned primitive data types into:
• 1D arrays • 2D arrays
is also supported. These will be 0-indexed.

➢ All the variables are signed by default. But we can declare an unsigned version of integers by prefixing the data type keyword with a '**u**' as described in section 4.1.
➢ No implicit type casting is performed.

2. Operations

This language supports the following operations with the respective associativity rules. They are listed in decreasing order of preference:

Type	**Operation**	**Symbol**	**Associativity**
Logical / Bitwise	NOT	**!**	R2L
Arithmetic	Modulo, Multiplication, Division	**%** ***** **/**	L2R
Arithmetic	Addition, Subtraction	**+** **-**	L2R
Relational	Less than, Less than equal to	**<** **<=**	L2R
Relational	Greater than, Greater than equal to	**>** **>=**	L2R
Relational	Equal to, Not equal to	**==** **!=**	L2R

Bitwise	AND	**&**	L2R
Bitwise	OR	**|**	L2R
Logical	AND	**&&**	L2R
Logical	OR	**||**	L2R
Assignment	Assignment	**=**	R2L

3. Semantics

- Type compatibility in operations:
  - Arithmetic can be performed on only **unsigned/signed integers** to result in an **unsigned/signed integer**.
  - Relational operators can be used with all primitive data types and results in the respective primitive data type.
  - Logical operators can be used with only **bool** and result in **bool.**
  - Bitwise operators can be used with **unsigned/signed integers** and **bool** to result in **unsigned/signed integer** and **bool** respectively.
  - Assignment is only possible when the LHS and RHS are of the same type.
  - **!** can be used only with **bool** to result in **bool.**
  - **-** can be used only with **signed integer** to result in **signed integer.**

- Every identifier (variable or function) must be declared before it is used.

- No two identifiers must share the same name in the same innermost scope.

- The parameters of a function are considered to lie inside the scope of the block of the function.

- Values being returned by functions should match the return type of the function.

- Values being passed to a function call must conform with the data types in the function declaration.

- The conditions of **if-elif-else**, **while**, **for, conditional expression** must have **bool** data type.

- The two return values in a **conditional expression** must be of the same type.

- ➢ The sizes/indices of arrays should be of type **integer.**
- ➢ Every **return** statement must have a function that encapsulates it.
- ➢ Every **break/continue** statement must have a loop that encapsulates it (a function should not encapsulate these statements before a loop does).

4. Syntax

- ➢ Any program written in this language is completely encapsulated in the **Main{}** block.
- ➢ All non terminal symbols are presented in **bold**.

4.1. Micro Syntax

<INT>	->	**-**?[**0-9**]+
<UINT>	->	[**0-9**]+
<CHAR>	->	‘[**\u0000-\u0256**]?’
<BOOL>	->	(**true** | **false**)
<STR>	->	“ (~(” | ‘))* ”
<a_op1>	->	(* | **/** | **%**)
<a_op2>	->	(**+** | **-**)
<r_op1>	->	(**<** | **<=**)
<r_op2>	->	(**>** | **>=**)
<r_op3>	->	(**==** | **!=**)
<l_op1>	->	**&&**
<l_op2>	->	**||**
<b_op1>	->	**&**
<b_op2>	->	**|**
<assgn_op>	->	**=**

```
<un_op>         ->   ( ! | - )

<type>          ->   ( int | uint  | char | bool )

<ID>            ->   [_a-zA-Z][_a-zA-Z0-9]*
```

4.2. Macro Syntax

```
<prog>          ->   Main{ <stmt>* } EOF

<stmt>          ->   <var_decl> ;
                     |  <func_decl>
                     |  <var_assgn> ;
                     |  if( <expr> ) <block> [ elif( <expr> ) <block> ]* [else <block>]?
                     |  for( [<var_assgn> [, <var_assgn>]*]? ; <expr>? ;
                     [<var_assgn> [, <var_assgn>]*]? ) <block>
                     |  while(<expr>?) <block>
                     |  <block>
                     |  break ;
                     |  continue ;
                     |  return <expr>? ;
                     |  <func_call>
                     |  input [-> <var>]+ ;
                     |  output [<- (<var> | <CHAR> | <STR>) ]+ ;


<var_assgn>     ->   <var> <assgn_op> [ <expr> | <var_assgn> ]

<var_decl>      ->   <type> <var> [,  <var> ]*

<var>           ->   <ID> | <ID> [ <expr> ] | <ID> [ <expr> ][ <expr> ]

<func_decl>     ->   <type> <ID> ( [<type> <var> [, <type> <var>]*  ]? ) <block>

<func_call>     ->   <ID> ( [<expr> [, <expr>]*  ]? ) ;

<block>         ->   { [<stmt>]* }

<expr>          ->   <var>
                     |  <value>
                     |  <un_op> <expr>
                     |  <expr> <a_op1> <expr>
                     |  <expr> <a_op2> <expr>
                     |  <expr> <r_op1> <expr>
```

```
                              |  <expr> <r_op2> <expr>
                              |  <expr> <r_op3> <expr>
                              |  <expr> <b_op1> <expr>
                              |  <expr> <b_op2> <expr>
                              |  <expr> <l_op1> <expr>
                              |  <expr> <l_op2> <expr>
                              |  ( <expr> )
                              |  <func_call>
                              |  <expr> ? <expr> : <expr>

<un_op>              ->       ! | -

<value>              ->       <INT> | <UINT> | <CHAR> | <BOOL>
```