NIMIC

Preface
This is a reference manual for the NIMIC programming language as implemented by the NIMS

Lexical Tokens

- Keywords
- Identifiers
- Constant
- Operators

Keywords

int	uint	char
bool	for	while
break	continue	if
else	return	

Identifiers

Identifiers are used to give a name to an object. It begins with an underscore character _ or any letter, and is followed by any letter, digit or underscore character. Since Identifiers are case sensitive, so var and VAR don't specify the same object.

Constant

Constant is used to describe a literal numeric or character value. All constants are of a particular data type (int, uint, char, bool).

Operators

%	+	-	*	/
~	||	&&	==	!=
<	<=	>	>=	?
:	;	[	]	(
)	{	}		

Lexical Considerations

- In NIMIC, lexical tokens consist of identifiers, keywords, constants and operators.
- All NIMIC keywords are lowercase. Also keywords and identifiers are case sensitive.
- Operator is used to describe an operation applied to one or several objects.
- Comments are the part of the program which are not meaningful for the compiler but important for program readability and understanding. In NIMIC, comments start with "//" or "#" and ends with end-of-line
- Whitespace is one or more space/tab/line-break/comment-block
- Expressions end with ";"
- Keywords/Identifiers must be whitespace separated(not required for ";"), example a = b + c is correct but a = b+c or a=b+c or a=b + c are not
- Character Encoding : ASCII
- int is 32 bit signed and uint is 32 bit unsigned

Scope

NIMIC provides support only for 2 environments, namely global and method. All identifiers defined in the global environment can be used by the whole program. Identifier declared inside a method is restricted to that method only. Same Identifier can't be declared twice but method environment identifiers can shadow global identifiers. Everytime method is called it's considered a new environment.

Semantics

1. Program consists of field declarations and method declarations
2. field declaration is declaration of an globally accessible identifier/variable
3. method declaration is declaration of a function/procedure

Declarations

Simple

1. int : The 32-bit signed int data type can hold integer values in the range of -2,147,483,648 to 2,147,483,647. Globally instantiated as 0
   Example: int a, int b
2. uint : The 32-bit unsigned int data type can hold integer values in the range of 0 to 4,294,967,295. Globally instantiated as 0
   Example: uint a, uint b
3. char : The 8-bit signed char data type can hold integer values in the range of -128 to 127. Stores string literals. Globally instantiated as '\0'
   Example: char a,char b
4. bool : 2-bit data type, stores 'true' or 'false'. Globally instantiated as false
   Example: bool a

Complex

1. array : collection of several objects of same type
   global instantiation will be same as than of type of object used to declare
   Example: int a[10], char b[10]

Block

Block refers to a list of statements enclosed within curly braces '{' statement1; statement2; '}'. It is used to form the body of functions or control statements.

Expressions and Operators

An expression, denoted by <*expr*>, consists of at least one operand and zero or more operators. Operands can be of any type such as constants, variables or function calls that return some values.

Arithmetic Operators

Supported arithmetic operations: addition, subtraction, multiplication, division, along with modular division and negation.

- + : Adds the two operands together.
  Example: 2+4
- - : Subtracts the two operands together.
  Example: 4-2
- * : Multiply the two operands together.
  Example: 4*2
- / : Divides the two operands together.
  Example: 4/2
- % : Gives the remainder when the left operand is divided by the right operand.
  Example: 4%3

Assignment Statements

<*location*> and <*expr*> must have the same type. <*location*> = <*expr*> evaluates <*expr*> and copies it to location. <*location*> += <*expr*> evaluates <*expr*> and increments stored <*location*> value by <*expr*> value. The same is also true for *, /, and -. For array types, <*location*> and <*expr*> must refer to a single array element which is also a scalar value and treated accordingly.
Example: x+=2 ; x-=2; x/=2; x*=2;

Comparison Operators

These are used to compare 2 operands. Supported comparison Operator are > < == <= >= !=.
Example: x==y; 5>4; 3<1

Conditional Expression

The expressions return truth value when 2 operands are compared with comparison operators.

Ternary Operator:
It is a conditional statement with structure as follows:

Syntax : <*conditional-expr*> *? operand1 : operand2*

where if <*conditional-expr*> is evaluated to be true then *operand1* is executed, else *operand2* is executed.

Control Statements

1. While loop: The while statement is a loop statement that runs til the condition given returns true.

   Syntax : while(conditional-expression) block

   Example : while(x < 13) { x += 6; }
2. For loop : The for statement is a loop statement. The structure of the for statement contains three parts: variable initialization, conditional-expression (testing), and variable modification. It is very convenient for making counter-controlled loops.

   Syntax : for(assignment-expression; conditional-expression(testing) ; assignment-expression) block

   Example for(int i = 4; i < 10; i += 2) { output(i); }
3. if/ if-else: The if/if-else statement is used to conditionally execute the particular branch of the program, based on the truth value of a given <conditional-expression>. The only difference between if-else and if is that the later doesn't execute any code if the truth value of a given <conditional-expression> comes out to be false.

   Syntax : if(conditional-expression) block else block

   Example : if(x == 1) { x = 4;}

I/O Statements

- The inbuilt input(<identifier>) function is used to take input, where the user has to specify the format of the identifier and the identifier which needs to be inputted.

  Syntax : input(x);
- The inbuilt output(<identifier>) function is used to display output, where the user has to output the format of the identifier and the identifier which needs to be outputted, when no identifier is provided, a new line is displayed.

  Syntax : output(x);

Return Statement

The return statement is used to end the execution of a function and return to the calling function(function that called that particular program).

Syntax : return <expr>

Functions

The functions are used to separate parts of your program into distinct subprocedures. The scope of the variable declared inside a function is limited to that function only. At the time of referencing a variable inside the function, local variables are used before global variables. A function can be called in the definition of function itself (recursion). All the function definitions(except main) must be declared at the start of the program. All the attributes of the function are assigned the value of the expression written in the declaration list at the time of calling the function. All attributes are treated as local variables in the scope of the function.
Syntax : return-type <name>(declaration-list) block
Example : uint func(uint attr1, char attr2) { uint x = attr1; return x; }

Semantic Checks

1. No identifier is declared twice in the same scope
2. The break and continue statements must be contained within the body of for or while statements
3. The number and type of arguments passed in the function call must be the/ same as at the time of defining/declaring the function.
4. The expression in a return statement must have the same type as the declared result type of the enclosing function definition.
5. All methods and identifiers must be declared before use
6. All identifiers must have type assigned to them at declaration
7. All methods must have type assigned, i.e, input arguments and return argument must have type assigned
8. There must be a main function, execution of program starts from there(after considering declarations)

Micro Syntax

identifier => [a-zA-Z_][a-zA-Z0-9_]*
alphabet => '[a-zA-Z]'
number => 0|[1-9][0-9]*

Macro Syntax

Notation

<NT>	: non terminal
<T>	: terminal
|	: or
{...}	: block for grouping(for or(s))
''	: no whitespace
{<>}*	: Zero or more occurrence
{<>}+	:One or more occurrence
e	: empty

<program> => {<field declarations>}* {<method declarations>}+ {<method blocks>}
<field declarations> => <type>{< id >',' <array id> }*<id>| <array id> ;
<type> => 'int' | 'uint' | 'char' | 'bool'
<array id> => <id> { '[' <int literal> ']' }+

<method declaration> => <type> <id> '(' { <type> ','}* <type> | e ')' ';'
<method block> => <type> <id> '(' { <type> <id> ','}* <type> <id> | e ')' '{' <block> '}'
<block> => {<field declarations>}* {<statements>}*

<statement> => {<id> | <id> '[' <expression> ']'} <assignment operator>
<expression> ';' | <method call> ';' | 'return' <expression> ';' | <conditionals> | <loops> |
'break' ';' |'continue' ';'

<method call> => <id> '(' {<expression> ','}* <expression> | e ')' ';'
<conditionals> => if '(' <expression> <comparison operator> <expression> ')' '{' <block>
'}' {else '{' <block> '}' | e} | <expression> <comparison operator> <expression> '?' <statement> :
<statement>

<loops> => 'while' '(' <condition> ')' '{' <block> '}' | 'for' '(' <assignment> ';'
<condition> ';' <assignment> ';')' '{' <block> '}'

<assignment> => <id> <assignment operator> <expression>
<condition> => <expression> <comparison operator> <expression>

<comparison operator> => '==' | '!=' | '<' | '>' | '<=' | '>='
<arithmetic operator> => '+' | '-' | '/' | '*' | '%'
<logical operator> => '&' | '|' |'~'
<assignment operator> => '=' | '+=' | '-=' | '/=' | '*='

<expression> => <literal> | <method call> | <id> | <expression> {<logical operator> |
<arithmetic operator>} <expression>| <io>

<literal> => <integer literal> | <unsigned integer literal> | <string literal> | <boolean literal>

<integer literal> => {signed 32 bit number}
<unsigned integer literal> => {unsigned 32 bit number}
<boolean literal> => 'true' | 'false'
<string literal> => {<alphabet>}+
<character> => {ASCII}
<io> => output<literal>| input<literal>