Ingenium Language Manual

K. Kaushik and P.Pradeep

August 23, 2019

As a part of Compilers course, we have designed a programming language called *Ingenium*, for which the language manual is presented here. We are inspired by c and c++ languages and designed our language in a way it reflects both these languages.

Requirements

We designed own programming language which supports the following features.

- Data Types: Signed and Unsigned integers, char, bool, 1D and 2D arrays.
- Arithmetic Operations: add, sub, mul, div, modulo
- Boolean Operations: and, or, not
- Control Statements: if-then, if-then-else, for loop, while loop, cond?opt1:opt2, break, continue
- Functions: Call by Value parameter passing mechanism, recursion support
- I/O Routines

Additional Features

1. Break statement which is similar to the break statement in C, Python Language. So, when a line of code has break statement the entire code flow would exit from the innermost while/for loop.
2. Continue statement which is similar to the continue statement in c.
3. Modulo also is one of the arithmetic operation which returns the remainder when a number is divided with the other.

Lexical Considerations

- All the keywords in **Ingenium** are lowercase. And any version of a keyword, that is not in the lower case is treated as a variable. Variable names are also case-sensitive.
- The keywords used are: `func, if, elsif, else, while, for, bool, int, unint, list, and, or, not, break, continue`
- Comments are represented using `//` for single line. Multi-line comments are enclosed in between `/*` and `*/` just like c. Comments are for human understanding and are not processed by the compiler. We use `"\n"` to denote a new line and `"\t"` to denote a literal tab.
- Variables should start with an alphabet following which can be any combination of numbers and letters. Any space indicates the end of the variable.
- `+, - , * , /, %` are the arithmetic operations add, subtract, multiply, divide and modulo operation respectively.
- Keywords `and, or, not` are the Boolean operators.
- I/O Routines : `scanin` and `printout` for stdio. `readin` and `writeout` to read from files and to write to files respectively.

Reference Grammar

Meta-notation :

$\langle x \rangle$	means x is a non terminal symbol.
x	(in bold font means that **x** is a terminal symbol, i.e; a token or a part of a token
$\left[\text{ x } \right]$	means zero or one occurrence of x, i.e, x is optional
x^*	means zero or more occurrence of x
$\text{x}^+,$	means a comma-separated list of one or more x's
|	denotes separate alternatives.
$\left\{ \right.$ $\left. \right\}$	for grouping, not to be confused with {}, which are part of the micro syntax of the language
a-z, A-Z, 0-9	Ranges of lower case, upper case and single digit numbers respectively

$\langle file \rangle \rightarrow \langle globaldecs \rangle \ \langle functions \rangle$

$\langle globaldecs \rangle \rightarrow \left\{ \textbf{list} \ \langle vartype \rangle \ \langle arrayname \rangle^+, \mid \langle vartype \rangle \left\{ \langle varname \rangle \left[= \langle expression \rangle \right] \right\}^+, \right\}^*;$

$\langle arrayname \rangle \rightarrow \ \langle varname \rangle \ [\langle uintliteral \rangle] \mid \langle varname \rangle \ [\langle uintliteral \rangle][\langle uintliteral \rangle]$

$\langle vartype \rangle \rightarrow \ \textbf{int} \mid \textbf{uint} \mid \textbf{bool} \mid \textbf{char}$

$\langle varname \rangle \rightarrow \ \left\{ a-z, A-Z \right\} \left\{ a-z, A-Z, 0-9 \right\}^*$

$\langle functions \rangle \rightarrow \left\{ \textbf{func } \langle funcName \rangle (\langle arglist \rangle) \ \textbf{\{} \left\{ \langle code \rangle \right\}^* \textbf{\}} \right\}^*$

$\langle arglist \rangle \rightarrow \left\{ \langle vartype \rangle \ \langle varname \rangle \right\}^*,$

$\langle funcName \rangle \rightarrow \left\{ a-z, A-Z \right\} \left\{ a-z, A-Z, 0-9 \right\}^*$

$\langle code \rangle \rightarrow \langle vardec \rangle \mid \langle loop \rangle \mid \langle ifdec \rangle \mid \langle statement \rangle \mid \langle methodcall \rangle \mid \langle returnblock \rangle \mid \langle multipleassign \rangle \mid \epsilon$

$\langle vardec \rangle \rightarrow \textbf{array } \langle vartype \rangle \ \langle arrayname \rangle^+ \textbf{, ;} \mid \langle vartype \rangle \left\{ \langle varname \rangle = \langle expression \rangle \right\}^+ \textbf{, ;}$

$\langle loop \rangle \rightarrow \langle forloop \rangle \mid \langle whileloop \rangle$

$\langle forloop \rangle \rightarrow \textbf{for (} \langle declaration \rangle \textbf{ ;} \langle condition \rangle \textbf{ ;} \langle expression \rangle \textbf{)\{} \langle code \rangle \textbf{\}}$

$\langle whileloop \rangle \rightarrow \textbf{while (} \langle condition \rangle \textbf{)\{} \langle code \rangle \textbf{\}}$

$\langle declaration \rangle \rightarrow \langle varname \rangle \ = \ \langle expression \rangle$

$\langle expression \rangle \rightarrow \langle location \rangle \mid \langle methodcall \rangle \mid \langle literal \rangle \mid \langle expression \rangle \ \langle binaryop \rangle \ \langle expression \rangle$
$\mid \langle ternaryop \rangle \mid \textbf{-} \ \langle expression \rangle \mid \textbf{not } \langle expression \rangle \mid \textbf{(} \langle expression \rangle \textbf{)}$

$\langle location \rangle \rightarrow \langle varname \rangle \mid \langle varname \rangle \ [\langle expression \rangle] \mid \langle varname \rangle \ [\langle expression \rangle][\langle expression \rangle]$

$\langle condition \rangle \rightarrow \epsilon \mid \langle expression \rangle \ \langle comparisonop \rangle \ \langle expression \rangle \mid \textbf{not } \langle expression \rangle$

$\langle comparisonop \rangle \rightarrow \textbf{!=} \mid \textbf{==} \mid \textbf{<=} \mid \textbf{>=} \mid \textbf{<} \mid \textbf{>}$

$\langle ifdec \rangle \rightarrow \textbf{if (} \langle condition \rangle \textbf{)\{} \langle code \rangle \textbf{\}} \left\{ \textbf{elsif (} \langle condition \rangle \textbf{)\{} \langle code \rangle \textbf{\}} \right\}^* \left[\textbf{else\{} \langle code \rangle \textbf{\}} \right]$

$\langle literal \rangle \rightarrow \mid \langle intliteral \rangle \mid \langle uintliteral \rangle \mid \langle boolliteral \rangle \mid \langle charliteral \rangle$

$\langle intliteral \rangle \rightarrow \left[\textbf{-} \right] \left\{ 0-9 \right\} \left\{ 0-9 \right\}^*$

$\langle uintliteral \rangle \rightarrow \left\{ 0-9 \right\} \left\{ 0-9 \right\}^*$

$\langle charliteral \rangle \rightarrow$ '$\langle char \rangle$'

$\langle boolliteral \rangle \rightarrow \textbf{true} \mid \textbf{false}$

$\langle ternaryop \rangle \rightarrow \langle condition \rangle \textbf{ ?} \ \langle expression \rangle \textbf{ :} \ \langle expression \rangle$

$\langle binaryop \rangle \rightarrow \langle arithop \rangle \mid \langle comparisionop \rangle \mid \langle booleanop \rangle$

$\langle arithop \rangle \rightarrow + \mid \textbf{-} \mid \textbf{*} \mid / \mid \%$

$\langle booleanop \rangle \rightarrow \textbf{true} \mid \textbf{false}$

$\langle statement \rangle \rightarrow \langle location \rangle = \langle expression \rangle \ ;$

$\langle methodcall \rangle \rightarrow \langle funcName \rangle \ \textbf{(} \ \langle arglist \rangle \ \textbf{)} \ ;$

⟨*returnblock*⟩ → **return** ⟨*varname*⟩ | **return** ⟨*intliteral*⟩;

Semantic Considerations

Each file in our Ingenium, first consists of global declarations, followed by a set of function declarations among them, a function called **main** is declared at last.

Global Declarations

Global declarations are a set of initialization of either arrays or simple variables. The syntax for these declarations are mentioned in a later section. Global declarations are accessible by all functions.

Function Declarations

Each file in out programming language should have a function called **main** which is executed first. The exact syntax of a function is specified in a later section.

Variable Naming and Types

A simple variable in *Ingenium*, is initialized with variable type followed by variable name and can be assigned a value. Assignment is permitted only for simple variables but not arrays.

An array variable is declared with the key word **list** followed by variable name, followed by an integer enclosed in [ ].

In *Ingenium*, variable names are atleast one character long, the first character is an English alphabet (lower and upper case, both) followed by any combination of alphanumeric characters.

Functions

Functions in *Ingenium*, are initialized with the keyword **func** followed by the name of the function,input parameters, code in order.

In *Ingenium*, function names are atleast one character long, the first character is an English alphabet (lower and upper case, both) followed by any combination of alphanumeric characters.

After the name of the function, the input parameters are enclosed in **( )**. The input parameters are a comma seperated list of variable type and variable name. Note that variable type can also include the key word **list**, indicating that we are passing an array of variables of certain type.

After the input parameters, the code to be executed when the function is invoked, is enclosed in **{}**. For returning the variable a key word called **return** should be used followed by the variable name which you want to return along with a semicolon. The type of the variable being returned and the type of the variable into which the output of the function is put into, should have the same type.

After syntactic and semantic checks, the function **main** is run first, which can invoke any number of other functions. All the functions should be declared above the main. If a program has no function named **main**, there will be no further execution of the program.

Data Types

Ingenium has **int**, **bool**, **uint**, **char**, and their respective 1D, 2D arrays.

int datatype

A variable of type **int**, can store the set of integers, positive, negative and 0. Negative numbers start with a **-** before the number.

bool datatype

A variable of type **bool**, takes only two values which are themselves key words, **true** and **false**.

uint datatype

A variable of type **uint**, store non-negative numbers. **int** and **uint** can be computed with a binary operator, but the **uint** variable is typecast into **int**, due to which it can be interpreted as a negative number also.

char datatype

A variable of type **char**, consists of ASCII symbols and store a single character.

list

list is same as the array in c.

The declaration of **1D** array follows this order, **list**,**datatype**,**variable name** then comes [**N**], where N is/evaluates an integer greater than 1. For example **list int a**[**10**], here we a have declared a 1D integer array.

The declaration of **2D** array follows this order, **list**,**datatype**,**variable name** then comes [**N**][**M**], where N,M is/evaluates an integer greater than 1. For example **list int a**[**10**][**10**], here we a have declared a 2D integer array.

In Ingenium, arrays can't be initialized. So, each element has to be initialized individually if needed.

I/O Routines

For performing stdio i.e. to read and write variables `scanin(<variable>)` and `printout(<variable>)` are used respectively.
While performing read and write from/to a file we use `readin(<filename/filepath>)` and `writeout(<filename/filepath>)` respectively.

Operators

Arithmetic Operators

+ , - , *, /, % are the arithmetic operations for addition, subtraction, multiplication, division and modulo operation respectively. None of these operations can be performed on variables of type