****

**Galaxy Programming Language**

GX is a syntax friendly and easy to understand language programming language without any ambiguity. One of the difference from other language is that GX using . (dot) for line breaking. GX License v1.0.

**Syntax of Galaxy**

- The default file extension for Galaxy files is ".gx".  
- GX statements end with a dot (.).  
- GX is a case sensitive language.   
- An equal sign (=) is used to assign, arithmetic operators ( + - \* / ) is used to compute values to variables.  
- Whitespace is required between keywords and identifiers.

- There is no declaration of variables, therefore implicitly specifying each data type for variables is not needed too.  
- Gx does not have a main function to start, it just starts to read from top to bottom, so it isn’t interpreted language.  
- Since it isn’t interpreted language, consts must have been declared at the top.

- Single line comments starts with **#,** multi line comments starts with **/\*** and ends with **\*/**.

**Tokens in Galaxy**

**BNF Form of Galaxy**

<program> ::= <stmt\_list>

<consts> ::= const <assignment> | const <assignment>. <consts>

<stmt\_list> ::= <consts> <stmt\_list>

| <function> <stmt\_list>

| <statement> <stmt\_list>

| <function>

| <statement>

<function> ::= <word> ( <parameters> ) start {<statement>}\* end

<line-breaker> ::= <dot>

<parameters> ::= <expr>

| <parameters> , <expr>

<expr> → <expr> or <and-expression>

| <and-expression>

| <equality-expression>

| <relational-expression>

| <additive-expression>

| <multiplicative-expression>

| <expression-with-head>

| <expression-with-tail>

<or-expression> ::= <and-expression>

| <or-expression> or <and-expression>

<and-expression> ::= <equality-expression>

| <and-expression> and <equality-expression>

<equality-expression> ::= <relational-expression>

| <equality-expression> == <relational-expression>

| <equality-expression> != <relational-expression>

<relational-expression> ::= <additive-expression>

| <relational-expression> < <additive-expression>

| <relational-expression> > <additive-expression>

| <relational-expression> <= <additive-expression>

| <relational-expression> >= <additive-expression>

<additive-expression> ::= <multiplicative-expression>

| <additive-expression> + <multiplicative-expression>

| <additive-expression> - <multiplicative-expression>

<multiplicative-expression> ::= <expression-with-head>

| <multiplicative-expression> \* <expression-with-head>

| <multiplicative-expression> / <expression-with-head>

| <multiplicative-expression> % <expression-with-head>

<expression-with-head> ::= <expression-with-tail>

| ++ <vars-and-consts>

| -- <vars-and-consts>

| not <expression-with-head>

<expression-with-tail> ::= <vars-and-consts>

| <expression-with-tail> ( {<parameters>}? )

| <expression-with-tail> <dot> <expression-with-tail>

| <expression-with-tail> ++

| <expression-with-tail> --

<expression-with-parenthesis> → (<expr>)

| <vars-and-consts>

<vars-and-consts> ::= <word>

| <string>

| ( <assignment> )

| <word> [ <assignment>]

<conditional-expression> ::= <expr> then <assignment> else <assignment>

<assignment> ::= <vars-and-consts> <assignment-operator> <expr>

| <vars-and-consts> <assignment-operator> <conditional-expression>

<statement> ::= {<assignment>}? <line-breaker>

| <condition>

| continue<line-breaker> | break<line-breaker> | return {<expr>}? <line-breaker>

| <lup>

<assignment-operator> ::= =

| \*=

| /=

| %=

| +=

| -=

<selection> ::= if <expr\_and\_statement> {elseif <expr\_and\_statement>}\*

{else <expr\_and\_statement>}? end

<expr\_and\_statement> :: = (<expr>) then <statement>

<lup> ::= while ( <expr> ) start <statement> end

| for ( <assignment><line-breaker> <expr> <line-breaker> <assignment> ) start <statement> end

**Contributors**

Eray Orçunus

Gökmen Özçelik

Caner Türkmen