GxScript 3.0 Language Reference

Yong He, http://www.csyong.net 2008/11/9

GxScript 3.0 Syntax Reference

Definitions and Conventions

There are two concepts that are important: terminals and non-terminals. Terminals are the end-point of parsers and cannot be derived from a terminal further. Basically, terminals are those characters or keywords that appear directly in the source code. Non terminals help represent the structure of a phrase and can be finally derived into terminals. The definition of non-terminals can be recursive.

In this document, all terminals and definitions of non-terminals are listed below. Terminals are indicated as blue bold fonts such as **for**, and non-terminals are all italic. The definition of non-terminals begins with the non-terminal name with bold and italic font and then with a colon(:). Alternative derivations are listed on separate lines.

Besides non-terminals and terminals, there are few symbols that helps simplify the derivation rules.

Symbol	Description
( )	Parenthesize a part of derivation rule for further indication
*****	Repeat the part before this symbol for zero or more times
+	Repeat the part before this symbol for one or more times
?	Indicates the part before this symbol is optional. It may repeat zero or one times
|	Indicates the part before this symbol can be replaced by the part after this symbol

Lexical syntax:

token:

identifier
int-literal
float- literal
string- literal
bool- literal
punctuator

identifier:

letter (*letter*|*digit*)*

letter: one of

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h I j k l m n o p q r s t u v w x y z _

digit: one of

1 2 3 4 5 6 7 8 9 0

bool-literal: one of

true false

int- literal:

digit+

float-literal:

digit (. digit*)?

string-literal:

“ string-content* **“**

string-content:

any Unicode character

escape-sequence

escape-sequence: one of

\” \t \s \n \r

keyword: one of

break case catch class const constructor continue default destructor do downto each else exception false for function handle host if in new operator overloads private property protected public return switch throw this to true try uses var while xor

operator: one of

[ ] ( ) . new + - * / % ^ && || xor & ! < > <= >= == != = += -= *= /= %= ^= ++ --

punctuator: one of

“ ; : , [ ] { } =

Phrase structure syntax

expression:

factor

unary-expression

exponential-expression

multiplicative-expression

additive-expression

comparative-expression

connect-expression

logic-and-expression

logic-or-expression

assignment-expression

unary-expression:

- factor
! factor

exponential-expression:

unary-expression
exponential-expression ^ factor

multiplicative-expression:

exponential-expression
*multiplicative-expression * exponential-expression*
multiplicative-expression / exponential-expression
multiplicative-expression % exponential-expression

additive-expression:

multiplicative-expression
additive-expression + multiplicative-expression
additive-expression – multiplicative-expression

connect-expression:

addive-expression
connect-expression & additive-expression

comparative-expression:

multiplicative-expression > multiplicative-expression
multiplicative-expression >= multiplicative-expression
multiplicative-expression < multiplicative-expression
multiplicative-expression <= multiplicative-expression
multiplicative-expression == multiplicative-expression
multiplicative-expression != multiplicative-expression

boolean-expression:

bool-literal
comparative-expression
logic-and-expression
logic-or-expression

logic-and-expression:

boolean-expression && boolean-expression

logic-or-expression:

boolean-expression **||** *boolean-expression*
boolean-expression **xor** *boolean-expression*

assignment-expression:

l-value **=** *expression*
l-value **+=** *expression*
l-value **-=** *expression*
l-value ***=** *expression*
l-value **/=** *expression*
l-value **%=** *expression*
l-value **^=** *expression*

l-value:

identifier
factor-index
factor-member
factor-call
comparative-expression
expression

factor:

identifier
int-literal
float-literal
bool-literal
string-literal
(*expression***)**
this
exception
new *class-name*(**(***argument-list***)**)?
function-definition
overload-definition
array-definition
factor-call
factor-index
factor-member
factor ++
factor –

argument-list:

expression

argument-list **,** expression

array-definition:

[*argument-list***]**

factor-call:

factor **(** *argument-list* **)**

factor-member:

*factor***.***identifier*

factor-index:

*factor***[***expression***]**

function-definition:

function (*parameter-list* **)** *statement-block*
overload-definition:
overloads {*function-definition-list***}**

function-definition-list:

function-definition
function-definition-list function-definition

parameter-list:

parameter
*parameter-list***,** *parameter*

parameter:

(*type-modifier*)? *identifier* (**=***expression*)?

type-modifier: one of

array bool float int string function numeral *class-name*

statement:

*expression***;**
statement-block
class-definition
ident-declaration
if-statement

for-statement
foreach-statement
while-statement
do-statement
jump-statement
switch-statement

statement-block:

{ *statement-list* **}**

statement-list:

statement
statement-list statement

class-definition:

class *class-name* (**:** *base-class-list*)? **{** *class-definition-statement-list* **}**

class-name:

identifier

base-class-list:

class-name
base-class-list, *class-name*

class-definition-statement-list:

class-definition-statement
class-definition-statement-list class-definition-statement

class-definition-statement:

(**public**|**protected**|**private** :)? *ident-declaration*
constructor (*parameter-list* **)** *statement-block*
destructor (*parameter-list* **)** *statement-block*
property *identifier* **{** (**set(***parameter***)** *statement-block*)? **get()** *statement-block* **}**
overload *operator* **(***parameter***)** *statement-list*

ident-declaration:

var|**const** *identifier* (**=***expression*)? **;**

if-statement:

if (*boolean-expression***)** *statement* (**else** *statement*)?

for-statement:

for (*expression***; *expression***; *expression***)** *statement*

foreach-statement:

for each *identifier* **in** *expression statement*

do-statement:

do *statement* **while(***boolean-expression***);**

while-statement:

while (*boolean-expression***)** *statement*

switch-statement:

switch (*expression* **) {** *case-list* **}**

switch-case:

case *expression***:** *statement*

default-case:

default: *statement*

case-list:

switch-case
case-list switch-case
case-list default-case

try-statement:

try *statement* **catch** *statement*

jump-statement:

break;
continue;
return *expression***;**
throw *expression***;**