Programming Languages and Translators

Language Reference Manual

ART: Animation Rendering Tool

Brett Jervey - baj2125
Gedion Metaferia - gym2103
Natan Kibret - nfk2105
Soul Joshi - srj2120

October 26, 2016

Table of Contents

1 Language Reference Manual

1.1 Lexical Conentions

An ART program consists of a single source file with function, method, shape, struct definitions and variable declaration. All programs must define a main function which serves as an entry point to the program.

1.1.1 Tokens

The language is composed of the following types of tokens: identifiers, keywords, literals, operators and other separators.

1.1.2 Comments

ART allows for both block and single line comments. The characters `/*` introduce a block comment, which terminates with the characters `*/`. Block comments do not nest. The characters `//` introduce a line comment which terminates at a new line character. Comments can not occur within character literals. Example below:

```
/* double line comment here
Double line comment continues here*/
// single line comment here
```

1.1.3 Identifiers

A valid identifier consists of any sequence of letters (an underscore counts as a letter) and digits. An identifier cannot begin with a digit. They can be of any length and case. Identifiers are case sensitive; in particular “abc“ is not the same as “Abc”.

1.1.4 Keywords

The language has the following reserved words that may not be used for any other purpose:

User Defined Structures:

```
struct
shape
```

Control Flows:

```
timeloop
frameloop
for
while
if
else
return
continue
break
```

Types:

`char`
`double`
`int`
`void`
`vect`

1.1.5 Literals

An integer literal can be one of the following:

- A decimal literal: a sequence of digits that does not begin with a zero.
- An octal literal: a sequence of digits that begins with zero and is composed of only the digits 0 to 7.
- A hexadecimal literal: `'0X'` or `'0x'` followed by a sequence of case insensitive hex digits.

Character literals can be one of the following:

- A printable character in single quotes. E.g: `'x'`
- One of the following escape sequences: `'\n', '\t', '\v', '\b', '\r', '\f', '\a', '\\', ''\?', '\` , '\'', '\"'`
- A backslash followed by 1,2 or 3 octal digits in single quotes: E.g: `'\0'`
- A backslash followed by the letter x and a sequence of hex digits in single quotes: E.g: `'\x7f'`

For the hex and octal escape sequences, the behavior is undefined if the resulting value exceeds that of the largest character.

A floating literal consists of an integer part, a decimal point, a fraction part, an `e` or `E`, and an optionally signed integer exponent. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the `e` and the exponent (not both) may be missing.

A vector literal is two floating literals separated by a comma enclosed by a matching set of angular brackets(`< >`). Any white space separating these components is ignored.

1.2 Meaning of Identifiers

Identifiers can be used to refer to functions, structures, members of structures and variables. All variables are automatic in scope with no ability to give them a static context.

A brief description of types:

1.2.1 Basic Types

Integers (`int`)

An integer is a 32-bit signed 2's complement series of digits with the maximum range of 2147483647.

Doubles (`double`)

A double is a 64-bit double precision number.

Characters (`char`)
: Characters occupy 8-bits and come from the ASCII set of characters.

Vector (`vec`)
: A vector is a tuple of two doubles. The components can be accessed with the indexing operator `'[]'`.

Void (`void`)
: The `void` type is used to declare a function that returns nothing.

1.2.2 Derived Types

All types, with the exception of `void`, can be used to define the following derived types:

Arrays
: Arrays are contiguous sequences of objects of a given type. They can be declared as: `<type-name>[]` or `<type-name>[size]`, where `size` is constant expression.

Structures (`struct`)
: A structure is a sequence of named members of various types and a set of associated member functions (methods).

Shapes (`shape`)
: Shapes which are structures that need to implement a `draw` method.

1.3 Conversion

Explicit type casting is not allowed in the language. And the only conversion that occurs is the promotion of an integer to the equivalent double value when an integer is provided where a double is expected. This includes arithmetic operations between int and double types and assignment of an int to a variable of type double.

1.4 Operators

The following operators are allowed in the language:

```
+     addition
-     subtraction
*     multiplication
/     division
%     modulo
<     less than
>     greater than
<=    less than or equal to
>=    greater than or equal to
==    equivalence
!=    inequality
=     assignment
```

```
+=     plus assignment
-=     subtraction assignment
/=     division assignment
*=     multiplication assignment
++     increment operator (prefix and postfix)
--     decrement operator (prefix and postfix)
+      unary plus
-      negation
!      logical not
|      logical OR
&&     logical AND
[]     subscript
::     scope
&      pass by reference
```

1.4.1 Arithmetic Operators

```
+      addition
-      subtraction
*      multiplication
/      division
%      modulo
```

ART supports the basic arithmetic operators: `addition`, `subtraction`, `multiplication`, `division` and `modulo` (remainder operator). In general, the left and right operands for the operators have to be the same type after `int` to `double` conversion if necessary. The exceptional case is for vector double multiplication and division. The operators evaluate to the same type as their operands.

Integers can be used with all the arithmetic operators. The `division` operator performs integer division (decimal truncated from result). The meaning of `modulo` operator for integers is such that: `a == ( a / b) * b + (a % b).`

Doubles can be used with all athermic operators expect `modulo`.

Vectors can used with addition and subtraction operators where the resulting vector is a componentwise sum/difference of the operands. Vectors can be multiplied with and divided by doubles with the same meaning as vector-scalar multiplication. The operations take the following form: `<scalar> * <vector> ,<vector> * <scalar>` and `<vector> / <scalar>`. The two forms for multiplication are equivalent.

All arithmetic operations are left-associative with `multiplication`, `division` and `modulo` having higher precedence than `addition` and `subtraction`.

1.4.2 Relational operators

```
<      less than
>      greater than
```

```
<=     less than or equal to
>=     greater than or equal to
```

The relational operators include: `less than, greater than, less than or equal to,` and `greater than or equal to`. These operators can only be applied to types `int` and `double`. Since there is no `boolean` type, the operators return `0` for false and `1` for true.

1.4.3 Equality Operators:

```
==     equivalence
!=     inequality
```

The equality operators include `equivalence` and `inequality`. They can be used with integers, doubles and vectors. Like relational operators they return `0` for false and `1` for true.

1.4.4 Assignment operator:

```
=      assignment
+=     plus assignment
-=     subtraction assignment
/=     division assignment
*=     multiplication assignment
```

The assignment operators are principally the basic `assignment` operator `=` and the compound assignment operators with the form `op=` where `op` is one of the arithmetic operators.

The basic `assignment` operator, when applied with a non-array object , stores the value of the right operand in the memory location corresponding to the left operand. This implies the left operand must be an expression that refers to an object in memory. Moreover, the left and right operands must be of the same type after the necessary promotions.

For an array variable, the assignment makes the left operand an alias for the array object in the right operand. What happens to the array previously assigned to the left operand is implementation defined.

The meaning of a compound assignment operator `'l op= r'` is the same as `'l = l op r'` but with the expression `'l'` evaluated only once. Moreover the operation `'l op r'` must be defined.

The assignment operators are right associative operators.

1.4.5 Increment/decrement operators:

```
++     increment operator (prefix and postfix)
--     decrement operator (prefix and postfix)
```

These operators are a shorthand form of the expression `x=x+1` or `x=x-1` but with the expression `x` evaluated only once. They both have prefix and postfix forms. The prefix form `++x` evaluates to the value of `x` after incrementing. The postfix form `x++` evaluates to the value of `x` before incrementing. The same holds true for prefix and postfix decrement. Neither of these expressions can be used on the left side of an assignment.

1.4.6 Unary Operators:

```
+     Unary Plus
-     negation
!     logical not
```

Includes `unary plus` , `negation` and `logical not`. `Unary plus` and `negation` can take `int`, `double` and `vec` types as operands. `Logical not` applies only to `int` types.

The `negation` operator is equivalent to multiplying by negative. For vectors, this implies component wise negation. The `unary plus` operator is a nop added for symmetry.

`Logical not` results in `0` for non-zero values and a `1` when applied to `0`.

1.4.7 Logical Operators:

```
|     logical OR
&&    logical AND
```

These are the `OR` and `AND` operators. The operators take in two `int` operands and return `0` or `1`. `OR` returns `0` if both operands are `0,` and `1` otherwise. `AND` returns `1` if both operands are `1` and `0` otherwise. Both operators are short-circuited. That implies operands are evaluated left to right and the rightmost operand is not evaluated if the result can be determined from the leftmost one.

1.4.8 Subscript Operator:

```
[]    subscript
```

The `subscript` operator is used to access individual objects in arrays and the components of vectors at the given index (eg. `x[5]` gets the object at index 5). The first object location is always zero and accessing an index higher than the number of objects in a given array(which can be any number) or vector(always size 2) results in an `out of bounds error`.

1.4.9 Scope Operator:

```
::    scope
```

The `scope` operator is only used for a member function (method) definition where the left operand is an identifier for a `struct` or `shape` type and the right is the method name.

1.4.10 Pass by Reference Operator:

```
&     pass by reference
```

The `pass by reference` is a special operator which is used only in the argument list of a function definition. When this operator is applied to a valid type, the argument passed is passed by reference rather than by value as ART is a pass-by-value default language. In this way, the argument is not copied but directly used by the function. This operator cannot be used anywhere else.

1.5 Functions

1.5.1 Function Definition

A basic function follows this format:

```
<type> <name>(<parameter list>opt)
{
       Function body
}
```

The `<type>` signifies the return type of the function which can be any of the basic types, `struct/shape` types and arrays of those types including arrays of array types. Functions that don't return a value are defined with return type 'void'. The name of the function needs to be unique.

The parameter list can contain any number of parameters in the form of `<type> <name>`, where type is any valid type excluding `void` and name is any valid identifier which is used by the function to access the argument's value.

The parameter can also have the `pass-by-reference` operator(which is appended to the type) which signals to the function to pass arguments by reference rather than by value (ART is by default a `pass-by-value` language). The `pass-by-reference` means that the argument is not copied but the argument's name acts as it alias. The `pass-by-reference` operator can only appear in the function definition and not the function call and only `lvalues` can be passed in as the argument for a `pass-by-reference` parameter.

```
void exampleByRef(int& x) { x =7; }
void exampleByValue(int x) { x = 10;}

int x =5;
exampleByValue(x);  // x  is unchanged
exampleByRef(x);    // x is now set to 7
exampleByValue(3);  // can do this
exampleByRef(3);    // compiler error
```

The function body is the actual code that is executed when a function call is performed. If a function has a non-void return type, it must have a return statement in its body. The expression in the return statement must match the return type.

1.5.2 Function Calls

Function calls are in the form of `<name>(<argument list>)` where `<name>` is a name of a function that has been previously defined. The length and types of arguments in the argument list of a function call must match exactly length and types of the parameter list in the function definition.

A function call (to non `void`) will evaluate to a value of the type declared in the definition. This value is a copy of the value of the expression in the corresponding return statement in the function body. A function call to a void function has no value or equivalently has value `void`.

1.5.3 Builtin-functions

The symbol `'#'` serves as a tag for builtin functions.

```
#drawpoint( vec, int)
```

Draw point takes a vector parameter that contains position of the point along with integer that represents the color given to the pixel and passes it to the animation renderer. The last 8 bits of the color argument hold the red value, the next 8 bits the green value and the following 8 bits hold the red value. The format can be condensed as `'0x00RRGGBB'`.

```
#add(shape)
```

Add takes one argument of a shape type and adds to the list of shapes to be drawn by the animation loops. The function must appear only within the outermost scope of the main function(cannot appear in nested scopes) and the shape must be either by a global variable or appear in the outermost scope of the main function.

```
#add{shape ...}
```

This version of add has the same restrictions as the single parameter version but can take in any number of shape arguments.

1.6 Structs and Shapes

1.6.1 Struct Definition

A structure definition follows this format:

```
struct <name>
{
 <type> <member names>;
  ...
}
```

The name of the `struct` must be unique and along with keyword `struct`, forms the type-declaration for that specific struct. .

```
struct point { int x; int y}

struct point pt1; // declares variable pt1 of type  struct point
```

The body of the `struct` contains any number of variable declarations with a type and a name and belong within the scope of the structure. A variable cannot be assigned a value in the struct definition.

1.6.2 Member Access

The way to access a variables and methods of struct is by using the post-fix dot notation expression as illustrated in the following example

```
struct point pt1; // variable pt1 with type struct point

pt1.x = 1; // variable x in instance of struct point pt1 has value of 1

pt1.y = 2; // variable y is set to 2
```

1.6.3 Defining Member Functions

A member function (method) is a function that belongs within the scope of a `struct` and is defined as:

```
<type> <struct-name>::<function name>(parameter-list)
{ function body}
```

Since a member function is in the scope of a `struct`, it can directly refer to the struct's variables in its body. It can also call other member functions. The member variables/functions referred to in a method body correspond to the member variables of the object on which the method is called. In other words the struct variable is an implicit argument to the member function.

Member function calls are written in the format of:

```
<variable of type struct>.<name of member function>(parameter-list).
```

Example:

```
struct point { int x; int y};

vec point::getPoint(){  vec temp; temp[0] = x ; temp[1] = y; return temp}

struct point pt1
pt1.x= 1;
Pt1.y=1;

pt1.getPoint(); // returns a vectors with components 1,2
```

1.6.4 Defining Constructors and Struct Initialization

A constructor is a special method that initializes and returns an instance of a structure. A constructor has the same name as the struct it returns and hence a constructor definition has no return type. This also implies that there is only one constructor as function overloading is not supported.

The body of constructor has access to the members of the struct like other methods. A constructor call creates a new object and the body of the constructor is executed with the newly created object provided as an implicit argument. The body of the constructor does not have a return statement.

A constructor is called as if it were a function that had the same name as the `struct`. A constructor call evaluates to a newly initialized object and can be used anywhere an expression of the struct type is legal . For example, it can be assigned to other struct variables, passed to functions/methods and returned from functions/methods.

```
point::point(int pt1, int pt2t) { x = pt1; y =pt2; }

struct point pt = point(5,6); // pt has its x variable set to 5 and y to 6
```

The other way to initialize a `struct` is to list the values in braces with the same number of listed values as fields in the structure being initialized

```
struct point { int x; int y;}

struct point pt1 = {1,2}; /* x and y in struct point are now 1 and 2
respectively */
```

If a variable in a `struct` is not initialized there is no guarantee to what the variable will contain as a value.

Structures can also be nested. The list initializers can be nested to initialize structs with nested structs.

```
struct rectangle { struct point top, bottom;}

struct rectangle r1 = { {1,2}, {3,4}}
```

Aside: List initializers can also be used to initialize arrays. The nested form of list initializes can be used to initialize arrays of arrays.

1.6.5 Shapes:

Shapes follow all the same conventions of structure but have the additional requirement of needing to have a draw member function defined.

```
shape circle{ vec center; double radius} // creates new shape circle
```

The `draw` member function dictates how the shape will be drawn when used in a `timeloop` or `frameloop`.

The `draw` function usually contains either the logic that creates the values that are passed into `#drawPoint` or calls the `draw` methods of its member shapes.

1.7 Statements

Statements are the basic units of executions. The following types of statements are defined:

expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement
builtin-statement

1.7.1 Expression Statements

These are statements of the form `<expression>;`. The value of the expression is evaluated and any side effects the expression may have takes effect before the next statement begins.

1.7.2 Compound Statements

Compound statements have the following form:

`{` *declaration-list*$_{opt}$ *statement-list*$_{opt}$ `}`

The form of compound statements implies that variables declarations have to come before any statements in blocks (as well as function bodies).

The variables that are defined in a block only exist and are accessible within the body of the block after the point in which they are defined. This is elaborated further in the scopes and declarations section.

1.7.3 Selection Statements

The selection statement has the following forms in the language:

selection-statement:
`if ( expression )` *statement*
`if ( expression )` *statement* `else` *statement*

`if else` statement:

The language supports if else statements as selection statements.

If the expression, which must be of type `int`, evaluates to a non-zero value, the first substatement (the `if` statement) is executed. The second substatement (the `else` statement) is executed if the expression is evaluated to zero. Nesting of the if else statements is also supported.

To resolve the dangling else ambiguity, the else is associated to the nearest `if`.

1.7.4 Iteration Statements

Iteration statements specify loops. Iteration statements have the following forms in the language:

iteration-statement:
`while (` *expression* `)` *statement*
`for (` *expression*$_{opt}$ `;` *expression*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*
`for (` *declaration* *expression*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*
`timeloop ( dt =` *expression* `; end =` *expression*`)` *statement*
`frameloop ( fps =` *expression* `; frames =` *expression*`)` *statement*

`while` statement:

The language supports `while` statements as iteration statements.

In the `while` statement, the expression specifies a test. The substatement is executed repeatedly as long as the value of the expression, which must be of type `int`, is not equal to `zero`. The test, including all side-effects of the expression, takes place before each execution of the statement.

`for` statement:

The language supports `for` statements as iteration statements.

In the first form of the `for` statement, the first expression, which can be of any type , is evaluated only once, and specifies initialization for the loop. The second expression, which must be of type `int`, specifies a test which is evaluated before each iteration of the loop. The for loop is terminated if the second expression evaluates to zero. The third expression, which can be of any type, specifies a re-initialization for the loop as it is evaluated at the end of each iteration. Typically, the third expression specifies an incrementation.

The second form simply substitutes the first expression for a declaration. The variables defined in the declaration are local to the loop scope.

The first form of the for statement is equivalent to:

```
expression1 ;
while ( expression2 )
{
        statement
        expression3 ;
}
```

The second form is equivalent to :

```
{
        declaration
        while ( expression2 )
        {
                statement
                expression3 ;
        }
}
```

Any of the three expressions (including the declaration) may be dropped. A dropped second expression makes the implied test equivalent to testing a non-zero constant, which results in an infinite loop

Time loop:

Time loop is on the the two animation specific control flow. In the first expression, the variable `dt`(which is treated as an integer) is assigned a value that represent the render period (in milliseconds) that the animation using. The second expression sets `end` (which is also treated as a integer) to a value representing the total time (in milliseconds) that the animation runs for. The statement that follows can use `dt` and `end` but cannot change `dt` or end or create a variable called

`dt` or `end`. However, outside the statement, variables with these names can be declared but are masked inside the time loop statement.

At the end of each iteration of the time loop, the runtime makes a call to the renderer to draw all the shapes.

The time loop cannot be nested and must appear only in the main function and within the main function only in the outermost scope.

Frame Loop:

Frame loop is the second animation specific control structure. The first expression sets the `int` variable `fps` to the number of frames that are rendered per second.The second expression sets the `int` variable frames to the total number of frames for the animation. Other than this, time loop and frame loop are equivalent.

1.7.5 Jump Statements:

Jump statements transfer control unconditionally. Jump statements have the following forms in the language:

jump-statement:
`continue ;`
`break ;`
`return` *expression*$_{opt}$ `;`

`continue` statement:

A `continue` statement may only appear in an iteration statement. It passes control to the loop-continuation portion of the enclosing iteration statement.

`break` statement:

A `break` statement may only appear in an iteration statement. It terminates the execution of the smallest enclosing iteration statement, and passes control to the statement following the terminated statement.

`return` statement:

A function returns to its caller by the return statement. If an expression is provided, the value of the expression is return to the caller of the function. The expression must match the type returned by the function in which it appears, the only exception being the case where an `int` is automatically promoted to a `double` to match the return type. An expression must be provided if the return type is not `void`. The no expression return statement corresponds to functions that have `void` return type. Falling off the end of the function is equivalent to a `return` statement with no expression.

1.8 Variable Declaration

Declarations follow the following format:

declaration:
    type-name init-declarator-list ;

init-declarator-list:
    init-declarator
    init-declarator-list , *init-declarator*

init-declarator:
    identifier
    identifier = *initializer*

Type name corresponds to a non-void type or arrays thereof. The following are examples of valid declarations:

```
int x, y = 3, z  = 3;
int[] z = {1, 2, 3} , w = {4, 3, 5};
int[4] b = {1,2,3,4} , f = {1,};
Int[][3] a = { {1,2,3}, {3,4,5}};
```

The declaration can optionally be initialized. The effect of uninitialized variable declarations depends on whether the declarations happens in local or global (outside functions). Local variables are left uninitialized while global variables are zeroed (at the byte level).

For array declaration, the size of the highest most dimension can be omitted but that requires the use of a full array initializer to be provided from which the dimension can be inferred.

Partial array initializers (ending with a comma) can be provided for arrays that have fully defined size. The parts of the array for which the partial initializer doesn't provide values are zeroed.

1.9 Program Structure:

An ART program consists of struct/shape definitions, method definitions, function definitions and global variable declarations.

The entry point for an ART program is the `main()` function and must be defined for all program. It must have return value `int` which is used to indicate program status to the calling environment. Animation control structures in an ART program can only be used in the outermost scope of the main function.

1.10 Scoping rules and Object Lifetimes:

A program will be kept in one source file. All regular functions, member functions, `structs` and shapes, are immediately accessible anywhere in the source file after being defined. The order of member function definitions does not matter. A function defined after a particular struct can call the member functions of the struct even if the member function definitions appear later in the source.

Variables declared outside of any function have global scope and are visible at any point in the source after their declaration (definition). This variables persist through the duration of the program.

Member functions (functions that are within the scope of a struct/shape) have access to all the member variables and member function of that struct/shape.

Variables declared within blocks (including function/method bodies) are visible throughout the body of the block. The lifetime of this variables is up to the point they go out of scope.

In particular, the scope of a parameter of a function is throughout the block defining the function and these variables only live during the function call. The exceptions are variables passed through reference which live outside the function. The second form of the for statement defines its own scope where variables declared in the for statement declaration are persistent and visible. Similarly timeloops and frameloops form their own scopes where the corresponding variables `dt`, `end`, `fps`, `frames` are persistent and visible.

Variables names in a nested scope can have the same name as variables/functions names in the outer scope. But the new names hide the old names. Otherwise, names in the nesting scope are accessible from the nested scopes.

1.11 Grammar

translation-unit:
    external-declaration$_{opt}$
    translation-unit external-declaration

external-declaration:
    function-definition
    method-definition
    declaration
    struct-or-shape-definition

method-definition:
    method-declarator `(` *parameter-list*$_{opt}$ `)` *compound-statement*

function-definition:
    function-declarator `(` *parameter-list*$_{opt}$ `)` *compound-statement*

declaration:
    type-name init-declarator-list `;`

init-declarator-list:
    init-declarator
    init-declarator-list `,` *init-declarator*

init-declarator:
    identifier
    identifier `=` *initializer*

struct-or-shape-specifier:
    struct-or-shape identifier

struct-or-shape-definition:
    struct-or-shape identifier `{` *struct-declaration-list* `}`

struct-or-shape:
    `struct`
    `shape`

struct-declaration-list:
    struct-declaration
    struct-declaration-list struct-declaration

struct-declaration:
    type-name struct-declarator-list `;`

struct-declarator-list:
    identifier
    struct-declarator-list `,` *identifier*

type-name:
    type-specifier abstract-declarator$_{opt}$

type-specifier: `one of`
    `void char int double vec` *struct-or-shape-specifier*

abstract-declarator:
    abstract-declarator$_{opt}$ `[` *constant-expression*$_{opt}$`]`

method-declarator:
    type-name$_{opt}$ *identifier*`::`*identifier*

function-declarator:
    type-name identifier

parameter-list:
    parameter-declaration
    parameter-list `,` *parameter-declaration*

parameter-declaration:
    type-name identifier
    type-name `&` *identifier*

initializer:
    expression
    `{` *initializer-list* `}`
    `{` *initializer-list* `,` `}`

initializer-list:
    initializer
    initializer-list `,` *initializer*

statement:
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement
builtin-statement

builtin-statement:
`#drawpoint (` *expression* `,` *expression* `) ;`
`#add (` *expression* `) ;`
`#add {` *argument-expression-list* `} ;`

expression-statement:
expression$_{opt}$ `;`

compound-statement:
`{` *declaration-list*$_{opt}$ *statement-list*$_{opt}$ `}`

declaration-list:
declaration
declaration-list declaration

statement-list:
statement
statement-list statement

selection-statement:
`if (` *expression* `)` *statement*
`if (` *expression* `)` *statement* `else` *statement*

iteration-statement:
`while (` *expression* `)` *statement*
`for (` *expression*$_{opt}$ `;` *expression*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*
`for (` *declaration* *expression*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*
`timeloop ( dt =` *expression* `; end =` *expression* `)` *statement*
`frameloop ( fps =` *expression* `; frames =` *expression* `)` *statement*

jump-statement:
`continue ;`
`break ;`
`return` *expression*$_{opt}$ `;`

expression:
conditional-expression
postfix-expression assignment-operator expression

assignment-operator: `one of`
`=  *=  /=  %=  +=  -=`

conditional-expression:
logical-OR-expression

logical-OR-expression `?` *expression* `:` *conditional-expression*

constant-expression:
conditional-expression

logical-OR-expression:
logical-AND-expression
logical-OR-expression `||` *logical-AND-expression*

logical-AND-expression:
equality-expression
logical-AND-expression `&&` *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:
prefix-expression
multiplicative-expression `*` *unary-expression*
multiplicative-expression `/` *unary-expression*
multiplicative-expression `%` *unary-expression*

prefix-expression:
unary-expression
`++` *unary-expression*
`--` *unary-expression*

unary-expression:
postfix-expression
unary-operator unary-expression

unary-operator: `one of`
`+` `-` `!`

postfix-expression:
primary-expression
postfix-expression `[` *expression* `]`

postfix-expression `(` *argument-expression-list*$_{opt}$ `)`
postfix-expression `.` *identifier*
postfix-expression `++`
postfix-expression `--`

primary-expression:
identifier
literal
`(` *expression* `)`

argument-expression-list:
expression
argument-expression-list`,` expression

literal:
integer-literal
character-literal
floating-literal
vector-literal