```
(s0, {) -> s1
(s0, }) -> s2
(s0, () -> s3
(s0, )) -> s4
(s0, [) -> s5
(s0, ]) -> s6
(s0, *) -> s7
(s0, %) -> s8
(s0, -) -> s9
(s0, +) -> s11
(s0, =) -> s13
(s0, <) -> s15
(s0, >) -> s17
(s0, !) -> s19
(s0, &) -> s21
(s0, |) -> s23
(s0, ,) -> s25
(s0, .) -> s26
(s0, ;) -> s31
(s0, /) -> s32
(s0, :) -> s40
(s0, #) -> s42
(s0, _abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ) -> s28
(s0, 1234567890) -> s30
(s0, """") -> s33
(s7, =) -> s45
(s9, -) -> s10
(s9, =) -> s44
(s9, >) -> s39
(s9, .) -> s27
(s9, 1234567890) -> s30
(s11, +) -> s12
(s11, =) -> s43
(s11, .) -> s27
(s11, 1234567890) -> s30
(s13, =) -> s14
(s15, =) -> s16
(s17, =) -> s18
(s19, =) -> s20
(s21, &) -> s22
(s23, |) -> s24
(s26, 1234567890) -> s27
(s27, 1234567890) -> s27
(s28, _abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ) -> s28
(s28, 1234567890) -> s29
(s29, _abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ) -> s28
(s29, 1234567890) -> s29
(s30, .) -> s27
(s30, 1234567890) -> s30
(s32, *) -> s36
(s32, =) -> s46
(s32, /) -> s35
(s33, {}()[]*%-+=<>!&|,.;/:#_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890\t @$'?\\\n\r) -> s33
(s33, """") -> s34
(s35, {}()[]*%-+=<>!&|,.;/:#_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890""""\t @$'?\\) -> s35
(s36, {}()[]%-+=<>!&|,.;/:#_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890""""\t @$'?\\\n\r) -> s36
(s36, *) -> s37
(s37, {}()[]*%-+=<>!&|,.;:#_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890""""\t @$'?\\\n\r) -> s36
(s37, /) -> s38
(s40, :) -> s41

I(s0)
F(s1, L_CU_BRACKET)
F(s2, R_CU_BRACKET)
F(s3, L_PAREN)
F(s4, R_PAREN)
```

```
F(s5, L_SQ_BRACKET)
F(s6, R_SQ_BRACKET)
F(s7, ASTERISK)
F(s8, MOD)
F(s9, MINUS)
F(s10, DECR)
F(s11, PLUS)
F(s12, INCR)
F(s13, ASSIGN)
F(s14, EQ)
F(s15, LT)
F(s16, LT_EQ)
F(s17, GT)
F(s18, GT_EQ)
F(s19, NOT)
F(s20, NOT_EQ)
F(s21, BITAND)
F(s22, AND)
F(s23, BITOR)
F(s24, OR)
F(s25, COMMA)
F(s26, PERIOD)
F(s27, FLOAT_LITERAL)
F(s28, ID)
F(s29, ID)
F(s30, INT_LITERAL)
F(s31, SEMICOLON)
F(s32, SLASH)
F(s34, STRING_LITERAL)
F(s35, LINE_COMMENT)
F(s38, BLOCK_COMMENT)
F(s39, ARROW)
F(s41, SCOPE_RESOLUTION)
F(s42, POUND)
F(s43, PLUS_ASSIGN)
F(s44, MINUS_ASSIGN)
F(s45, ASTERISK_ASSIGN)
F(s46, SLASH_ASSIGN)

T(L_CU_BRACKET, "{")
T(R_CU_BRACKET, "}")
T(L_PAREN, "(")
T(R_PAREN, ")")
T(L_SQ_BRACKET, "[")
T(L_SQ_BRACKET, "]")
T(ASTERISK, "*")
T(MOD, "%")
T(MINUS, "-")
T(DECR, "--")
T(PLUS, "+")
T(INCR, "++")
T(ASSIGN, "=")
T(EQ, "==")
T(LT, "<")
T(LT_EQ, "<=")
T(GT, ">")
T(GT_EQ, ">=")
T(NOT, "!")
T(NOT_EQ, "!=")
T(BITAND, "&")
T(AND, "&&")
T(BITOR, "|")
T(OR, "||")
T(COMMA, ",")
T(PERIOD, ".")
T(SEMICOLON, ";")
T(SLASH, "/")
```

```

ReservedWord(int, INT)
ReservedWord(integer, INTEGER)
ReservedWord(long, LONG)
ReservedWord(short, SHORT)
ReservedWord(byte, BYTE)
ReservedWord(float, FLOAT)
ReservedWord(double, DOUBLE)
ReservedWord(real, REAL)
ReservedWord(precision, PRECISION)
ReservedWord(fixed, FIXED)
ReservedWord(char, CHAR)
ReservedWord(character, CHARACTER)
ReservedWord(bool, BOOL)
ReservedWord(boolean, BOOLEAN)
ReservedWord(void, VOID)
ReservedWord(true, TRUE)
ReservedWord(false, FALSE)
ReservedWord(for, FOR)
ReservedWord(while, WHILE)
ReservedWord(do, DO)
ReservedWord(if, IF)
ReservedWord(else, ELSE)
ReservedWord(return, RETURN)
ReservedWord(class, CLASS)
ReservedWord(using, USING)
ReservedWord(namespace, NAMESPACE)
ReservedWord(include, INCLUDE)

Ignore(LINE_COMMENT)
Ignore(BLOCK_COMMENT)


include =:
[POUND] [INCLUDE] [STRING_LITERAL]

using-namespace =:
[USING] [NAMESPACE] <namespace-identifier> [SEMICOLON]

function-prototype =:
<function-return-type> <function-identifier> <function-proto-parameters> [SEMICOLON];


function-definition =:
<function-return-type> <function-identifier> <function-parameters> <function-body>

statement-list =:
<statement> <statement-list>?

statement =:
<expression-statement>|<for-loop>|<while-loop>|<decision>|<block>

expression-statement =:
<expression> [SEMICOLON]

while-loop =:
[WHILE] [L_PAREN] <while-condition> [R_PAREN] <while-body>

for-loop =:
[FOR] [L_PAREN] <for-init> [SEMICOLON] <for-condition> [SEMICOLON] <for-increment> [R_PAREN] <for-body>

decision =:
[IF] [L_PAREN] <expression> [R_PAREN] <decision-body> <decision-cases>? <decision-fallback>?

decision-cases =:
<decision-case> <decision-cases>?

```

```
decision-case =:
[ELSE] [IF] [L_PAREN] <expression> [R_PAREN] <decision-body>

decision-fallback =:
[ELSE] <decision-body>

decision-body =:
<block>|<statement>|[SEMICOLON]

block =:
[L_CU_BRACKET] <statement-list>? [R_CU_BRACKET]

member-access =:
<member-access-head> <member-access-tail>

method-invocation =:
<function-identifier> [L_PAREN] <arg-list>? [R_PAREN]

expression =:
<grouped-expression>|<method-invocation>|<declaration>|<assignment>|<operation>|<return>|<simple-expression>

simple-expression =:
<member-access>|<subscript-access>|<literal>|<identifier>

operation =:
<binary-expression>|<unary-expression>

subscript-access =:
<subscript-access-head> <object-subscript>

subscript-access-head =:
<method-invocation>|<identifier>

grouped-expression =:
[L_PAREN] <expression> [R_PAREN]

declaration =:
<type> <initializer-list>

initializer-list =:
<identifier> $DeclareVar([0], ("declaration")[0])$ <initializer-assignment-tail>? <initializer-list-tail>?

initializer-list-tail =:
[COMMA] <identifier> $DeclareVar([1], ("declaration")[0])$ <initializer-assignment-tail>? <initializer-list-tail>?

assignment =:
<assignment-target> <assignment-tail>

assignment-target =:
<member-access>|<setter>

binary-expression =:
<relational-expression>|<algebraic-expression>|<logical-expression>

relational-expression =:
<operation-expression> <relational-expression-tail>

algebraic-expression =:
<operation-expression> <algebraic-expression-tail>

logical-expression =:
<operation-expression> <logical-expression-tail>

operation-expression =:
<grouped-expression>|<literal>|<identifier>


```

```
relational-expression-tail =:
<relational-binary-op> <operation-expression> <relational-expression-tail>?

algebraic-expression-tail =:
<math-binary-op> <operation-expression> <algebraic-expression-tail>?

logical-expression-tail =:
<logical-binary-op> <operation-expression> <logical-expression-tail>?


unary-expression =:
<not-expression>|<unary-postfix-expression>|<unary-prefix-expression>

not-expression =:
[NOT] <identifier>

unary-postfix-expression =:
<identifier> <unary-op> $AccumulateVar([1], [0])$

unary-prefix-expression =:
<unary-op> <identifier> $AccumulateVar([0], [1])$

literal =:
<bool-literal>|[FLOAT_LITERAL]|[INT_LITERAL]|[STRING_LITERAL]

type =:
<primitive-type>|<identifier> <asterisk-tail>?

asterisk-tail =:
[ASTERISK] <asterisk-tail>?

primitive-type =:
<int-primitive>|<float-primitive>|<fixed-primitive>|<char-primitive>|<bool-primitive>|<void>

binary-op =:
<math-binary-op>|<logical-binary-op>|<relational-binary-op>

return =:
[RETURN] <expression>?

identifier =:
[ID]


void =:
[VOID]

int-primitive =:
[INT]|[LONG]|[SHORT]|[BYTE]|[INTEGER]

float-primitive =:
[FLOAT]|[DOUBLE]|[REAL]|<double-precision>

double-precision =:
[DOUBLE] [PRECISION]

fixed-primitive =:
[FIXED]

char-primitive =:
[CHAR]|[CHARACTER]

bool-primitive =:
[BOOL]|[BOOLEAN]

bool-literal =:
[TRUE]|[FALSE]
```

```

math-assign-op =:
[PLUS_ASSIGN]|[MINUS_ASSIGN]|[ASTERISK_ASSIGN]|[SLASH_ASSIGN]

math-binary-op =:
[PLUS]|[MINUS]|[ASTERISK]|[SLASH]

logical-binary-op =:
[BITAND]|[AND]|[BITOR]|[OR]

relational-binary-op =:
[EQ]|[LT]|[LT_EQ]|[GT]|[GT_EQ]|[NOT_EQ]

unary-op =:
[INCR]|[DECR]


declaration-list =:
<declaration> <declaration-list-tail>?

arg-list =:
<expression> <arg-list-tail>?

declaration-list-tail =:
[COMMA] <declaration> <declaration-list-tail>?

arg-list-tail =:
[COMMA] <expression> <arg-list-tail>?

member-access-head =:
<getter>

member-access-operator =:
[PERIOD]|[ARROW]

member-access-tail =:
<member-access-operator> <getter> <member-access-tail>?


object-subscript =:
[L_SQ_BRACKET] <expression> [R_SQ_BRACKET]

function-return-type =:
<type>

function-identifier =:
[ID]

function-proto-parameters =:
[L_PAREN] <function-proto-parameter-list>? [R_PAREN]

function-parameters =:
[L_PAREN] <function-parameter-list>? [R_PAREN]

function-parameter-list =:
<function-parameter-declaration> <function-parameter-list-tail>?

function-parameter-list-tail =:
[COMMA] <function-parameter-declaration> <function-parameter-list-tail>?

function-proto-parameter-list =:
<function-proto-parameter-declaration> <function-proto-parameter-list-tail>?

function-proto-parameter-list-tail =:
[COMMA] <function-proto-parameter-declaration> <function-proto-parameter-list-tail>?

function-body =:
```

```
<block>


function-parameter-declaration =:
<type> <identifier> $DeclareVar([1], [0])$ <function-parameter-assignment>?

function-proto-parameter-declaration =:
<type> <identifier>? <function-parameter-assignment>?

function-parameter-assignment =:
[ASSIGN] <assign-expression>

for-init =:
<expression>?

for-condition =:
<expression>?

for-increment =:
<expression>?

for-body =:
<block>|<statement>|[SEMICOLON]

while-condition =:
<expression>?

while-body =:
<block>|<statement>|[SEMICOLON]

assignment-tail =:
<algebraic-assignment-tail>|<standard-assignment-tail>

initializer-assignment-tail =:
[ASSIGN] <assign-expression> $AssignVar(^[@-1], ResolveExpr([1]))$

standard-assignment-tail =:
[ASSIGN] <assign-expression> $AssignVar(^^[@-1], ResolveExpr([1]))$

algebraic-assignment-tail =:
<math-assign-op> <assign-expression> $AccumulateVar([0][0], ^^[@-1], [1])$

assign-expression =:
<grouped-expression>|<method-invocation>|<assignment>|<operation>|<simple-expression>

namespace-identifier =:
[ID] <namespace-identifer-tail>?

namespace-identifier-tail =:
[SCOPE_RESOLUTION] [ID] <namespace-identifier-tail>?


getter =:
<method-invocation>|<subscript-access>|<identifier>

setter =:
<subscript-access>|<identifier>
```