KXI 2020
Note: There is NO string in KXI

Note: There are no floating point numbers or strings in KXI.

Meta-Language

```
::=    is defined as                   |     alternative definition
;      end of rule                     [ ]   optional
{ }    zero or more occurrences        ( )     grouping
 x       non-terminal symbol x         "x"     terminal symbol x
"\""     terminal symbol "
```

Comment

```
Comment ::= "//" comment until the end of the line

letter ::= Any ASCII character from "A" to "Z" or "a" to "z"

identifier ::= letter { letter | number } ;
Handle identifiers of at least length 21

character::=
          printable_ascii
        | nonprintable_ascii
        ;
```

Names, Types and Literals

```
keyword ::=
       "atoi" | "and" | "bool" | "block" | "break" | "case" | "class" |
"char" | "cin" | "cout" | "default" | "else" | "false" | "if" | "int" |
"itoa" | "kxi2020" | "lock" | "main"| "new" | "null" | "object" | "or"
| "public" | "private" | "protected" | "return" | "release" | "string"
| "spawn" | "sym" | "set"| "switch" | "this" | "true" | "thread" |
"unprotected" | "unlock" | "void" | "while" | "wait"
      ;

modifier::= "public" | "private"
        ;

class_name::= identifier ;


type ::=  "int" | "char" | "bool" | "void" | "sym" | class_name
      ;

character_literal::= "\'" character "\'" ;
These are tokens found by your lexical analysis.

numeric_literal::= ["+" | "-"]number ;
These are tokens found by your lexical analysis.


number::=
          "0"{number} | "1"{number} | "2"{number} | "3"{number}
        | "4"{number} | "5"{number} | "6"{number} | "7"{number}
        | "8"{number} | "9"{number}
        ;

printable_ascii::=
      These are the ASCII values between decimal 32 (SPACE) to 126 (~)
      found by your lexical analysis.


nonprintable_ascii::=
      Nonprintable ASCII values are between decimal 0 (null) to 31
      (unit separator) as well as 127 (DEL) found by your lexical
      analysis. They can be formed by combining a '\' with a printable
      ASCII character '\n', '\r','\t' and for example.
```

Case_Block

```
case_block::= "{" {case_label} "default" ":" statement "}"

case_label::= "case" literal ":" statement ;

literal::= numeric_literal | character_literal ;
```

Start Symbol

```
      compiliation_unit::=
            {class_declaration}
            "void" "kxi2020" "main" "(" ")" method_body
            ;
```

Declarations

```
class_declaration::=
      "class" class_name "{"
      {class_member_declaration} "}"
      ;

class_member_declaration::=
        modifier type ["[" "]"] identifier field_declaration
      | constructor_declaration
      ;

field_declaration::=
       ["=" assignment_expression ] ";"
      | "(" [parameter_list] ")" method_body
      ;

constructor_declaration::=
      class_name "(" [parameter_list] ")" method_body ;

method_body::=
      "{" {variable_declaration} {statement} "}" ;

variable_declaration::=
      type ["[" "]"] identifier ["=" assignment_expression ] ";" ;

parameter_list::= parameter { "," parameter } ;

parameter::= type ["[" "]"] identifier;
```

Statement

```
statement::=
        "{" {statement} "}"
      | expression ";"
      | "if" "(" expression ")" statement [ "else" statement ]
      | "while" "(" expression ")" statement
      | "return" [ expression ] ";"
      | "cout" "<<" expression ";"
      | "cin" ">>" expression ";"
      | "switch" "(" expression ")" case_block
      | "break" ";"
      ;
```

Expression

```
expression::=
        "(" expression ")" [ expressionz ]
      | "true" [ expressionz ]
      | "false" [ expressionz ]
      | "null" [ expressionz ]
      | "this" [ member_refz ] [ expressionz ]
      | numeric_literal [ expressionz ]
      | character_literal [ expressionz ]
      | identifier [ fn_arr_member ] [ member_refz ] [ expressionz ]
      ;

/* function or array member element */
fn_arr_member::= "(" [ argument_list ] ")" | "[" expression "]"
      ;

argument_list::= expression { "," expression } ;

/* reference a class member, can be a variable, function, or array */
member_refz::= "." identifier [ fn_arr_member ] [ member_refz ] ;


expressionz::=
        "=" assignment_expression
      | "and" expression            /* logical connective expression */
      | "or" expression       /* logical connective expression */
      | "==" expression       /* boolean expression */
      | "!=" expression       /* boolean expression */
      | "<=" expression       /* boolean expression */
      | ">=" expression       /* boolean expression */
      | "<" expression        /* boolean expression */
      | ">" expression        /* boolean expression */
      | "+" expression        /* mathematical expression */
      | "-" expression        /* mathematical expression */
      | "*" expression        /* mathematical expression */
      | "/" expression        /* mathematical expression */
      ;

/* assign either an expression, new class object or new array object */
```

```
assignment_expression::=
        expression
      | "new" type new_declaration
      ;

new_declaration::=
      "(" [ argument_list ] ")"
      | "[" expression "]"
      ;
```