```
program ::=
    exp
  | decs

exp ::=
  # Literals.
    nil
  | integer
  | string

  # Array and record creations.
  | type-id [ exp ] of exp
  | type-id {[ id = exp { , id = exp } ] }

  # Object creation.
  | new type-id

  # Variables, field, elements of an array.
  | lvalue

  # Function call.
  | id ( [ exp { , exp }] )

  # Method call.
  | lvalue . id ( [ exp { , exp }] )

  # Operations.
  | - exp
  | exp op exp
  | ( exps )

  # Assignment.
  | lvalue := exp

  # Control structures.
  | if exp then exp [else exp]
  | while exp do exp
  | for id := exp to exp do exp
  | break
  | let decs in exps end

lvalue ::= id
  | lvalue . id
  | lvalue [ exp ]
exps ::= [ exp { ; exp } ]

decs ::= { dec }
dec ::=
  # Type declaration.
    type id = ty
  # Class definition (alternative form).
  | class id [ extends type-id ] { classfields }
  # Variable declaration.
  | vardec
  # Function declaration.
  | function id ( tyfields ) [ : type-id ] = exp
  # Primitive declaration.
  | primitive id ( tyfields ) [ : type-id ]
  # Importing a set of declarations.
  | import string

vardec ::= var id [ : type-id ] := exp

classfields ::= { classfield }
# Class fields.
classfield ::=
```

```
  # Attribute declaration.
    vardec
  # Method declaration.
  | method id ( tyfields ) [ : type-id ] = exp

# Types.
ty ::=
   # Type alias.
     type-id
   # Record type definition.
   | { tyfields  }
   # Array type definition.
   | array of type-id
   # Class definition (canonical form).
   | class [ extends type-id ] { classfields }
tyfields ::= [ id : type-id { , id : type-id } ]
type-id ::= id

op ::= + | - | * | / | = | <> | > | < | >= | <= | & | |
Precedence of the op (high to low):

* /
+ -
>= <= = <> < >
&
|
Comparison operators (<, <=, =, <>, >, >=) are not associative. All the remaining
operators are left-associative.
```