Especificaciones para el Lenguaje MJ

Para el proyecto de ***Compiladores***, se debe construir un compilador para un lenguaje basado en objetos llamado MJ (para Mini Java) que es esencialmente una versión simplificada de Java. A pesar de todas las simplificaciones que nos permite tener una especificación de gramática de una página, el lenguaje conserva bastantes rasgos y construcciones de Java para hacer de este proyecto un desafío y permitirnos escribir muchos programas interesantes. La *figura 1* presenta la gramática completa del lenguaje. En el resto de este documento se da una descripción detallada, discutiendo su sintaxis y semántica.

Descripción

La siguiente lista resalta los rasgos principales del lenguaje:

- ***Características Orientas a Objetos***: soporta objetos, herencia simple, subtipos, y llamadas a métodos; sin embargo, no soporta sobrecarga de métodos;
- ***Tipos***: Es un lenguaje fuertemente tipado y proporciona: tipos primitivos para enteros, lógicos, y cadenas; tipo clase para objetos; y tipo para arreglos.
- ***Chequeos en tiempo de ejecución***: soporta chequeos en tiempo de ejecución para las referencias nulas, violaciones al límite de los arreglos, y para la asignación negativa del tamaño de los arreglos.

Consideraciones léxicas

Identificadores y palabras reservadas son sensibles a las mayúsculas (case-sensitive). Los identificadores pueden empezar con un carácter alfabético o el carácter subrayado (_). Siguiendo al carácter inicial puede haber cualquier secuencia de caracteres alfabéticos, caracteres numéricos, o el carácter de subrayado (_). Caracteres alfabéticos en mayúscula y en minúsculas son considerados alfabéticos y diferentes, así x y X son dos identificadores diferentes. Las siguientes son las palabras reservadas en el lenguaje y no pueden usarse como identificadores:

class	extends	void	int	boolean	string
return	if	else	while	break	continue
this	new	length	true	false	null

. Proponga una estrategia para las palabras reservadas.
. Un identificador no puede iniciar con un número ni con una ñ o las vocales tildadas
. Un identificador no puede terminar con un número
. Un identificador puede contener la letra ñ y las vocales tildadas (mayúsculas y minúsculas)
. Una vez reconocido el identificador, si éste contiene ñ o vocales tildadas, se debe reemplazar estos caracteres con un guión bajo. Ejemplos
Año A_o
presión presi_n

Espacio en blanco consiste en una sucesión de uno o más espacios, tab, o caracteres *newline*. Los espacios blancos pueden aparecer entre cualquier *componente léxico*. Las palabras reservadas e identificadores deben de estar separados por un espacio en blanco o un *componente léxico* que no sea

ni palabra clave o identificador. Por ejemplo, **elsex** representa un solo identificador, no la palabra clave *else* seguida por el identificador **x**.

comentarios de Java: Un comentario que empieza con los caracteres **//** indica que el resto de la línea es un comentario. Además, un comentario puede ser una sucesión de caracteres que empiezan con /*, seguido por cualquier carácter, incluyendo *newline*, hasta la primera ocurrencia de la secuencia de fin */. Comentarios sin cerrar que inician con la secuencia /*, pero no tiene una secuencia de cierre */, se consideran un error de léxico.

Literales enteros pueden empezar con un signo negativo opcional -, seguido por un 0, o un dígito en el rango 1-9 y una secuencia de cero o más dígitos en el rango 0-9. Los enteros son valores de 32-bits entre -2^{31} y $2^{31} - 1$ (es decir, valores enteros entre -2147483648 y 2147483647). Diferentes números pueden ser separados por un espacio blanco o un *componente léxico* que no sea ni palabra reservada o un identificador. **Números e identificadores también deben ser separados por espacios blancos u otros *componente léxico***.

Además, se implementará los números en formato científico, por ejemplo:
-2.34 E-2

Numeros binarios: Se utilizará la representación de números en formato Binario, con el siguiente formato: ***b'100001'***, para decir que la anterior secuencia es un número en formato Binario. Además, se debe almacenar el respectivo valor en decimal.

Cadenas de caracteres son secuencias de caracteres encerradas entre **comillas**. La secuencia de caracteres en una cadena consiste de:

1) Caracteres ASCII imprimibles (códigos ASCII decimales entre el 32 y 126), la comilla " y la barra inversa \.

2) **la secuencia de escape \" para denotar la comilla, \\ para denotar la barra inversa, \t para denotar tab, y \n para el newline. Ningún otro carácter o sucesiones del carácter pueden ocurrir en una cadena. Cadenas no cerradas, la cual inicia con una comilla ", pero no tiene una comilla de cierre ", es considerado un error de léxico.**

Las literales booleanas pueden ser una de las palabras reservadas true o false. La única literal para referencias es null.

Operadores

Operadores unarios y binarios incluyen:

- Operadores Aritméticos: adición +, substracción -, multiplicación *, división / y módulo %. Todos estos operadores solamente con enteros

- Operadores de comparación relacional: menor que <, menor o igual a <=, mayor que >, mayor o igual a >= Los operandos para operadores de relación deben ser expresiones enteras.

- Operadores de comparación de igualdad

- Operadores unarios: cambio de signo - para enteros y negación lógica ! para booleanos.

La tabla 1 muestra la precedencia y asociatividad de los operadores:

Prioridad	Operador	Descripción	Asociatividad
1	[] () .	índices arreglos, llamadas a métodos acceso a campos/métodos	izquierda
2	- ! new	menos unario, negación lógica asignación de objetos	derecha
3	* / %	multiplicación, división, módulo	izquierda
4	+ - +	adición, substracción concatenación cadena caracteres	izquierda
5	< <= > >=	operadores de relación	izquierda
6	== !=	comparación de igualdad	izquierda
7	&&	Y	izquierda
8	||	O	izquierda
9	=	asignación	derecha

Tabla 1. Precedencia y asociatividad de los operadores.

Estructura de un programa en minijava

Un programa consiste de una secuencia de declaraciones de clase. Cada clase en el programa tiene una secuencia de campos y declaración de métodos. Un programa debe contener un solo método **main**, con la siguiente forma:

```
void main ( string[] args) { ... }
```

El método main es dinámico. Esto es, si main pertenece a una clase **A,** la ejecución del programa consiste en crear un nuevo objeto de la clase A, luego ejecutar su método main. Observe que esto es incompatible con Java en el cual el método main es estático.

Variables del programa

Las variables de un programa pueden ser **locales o los parámetros** de un método, en cuyo caso deben ser localizadas en la pila (stack); o pueden ser campos del objeto o elementos de un arreglo, localizados en el montículo (heap). Las variables del programa del tipo int y boolean contendrán valores enteros y booleanos, respectivamente. Variables de otro tipo contienen referencia a estructuras localizadas en el montículo (p.e, cadenas, arreglos u objetos).

El programa inicializa las variables con sus **valores por defecto** cuando son declaradas. Enteros tendrán un valor por defecto de 0, boolean tendrán un valor de false y referencias tendrán un valor por defecto de null. Variables locales son inicializadas cuando el método es invocado. Campos de objetos y elementos de arreglos son inicializados cuando tales estructuras son dinámicamente creadas.

Cadenas

Para referenciar cadenas de caracteres, el lenguaje usa el tipo primitivo string. Este es diferente al de Java, donde String es una clase y así las cadenas de caracteres pueden ser extendidas por otra clase.

Las cadenas de caracteres son asignadas al montículo (heap) y son inmutables, lo cual significa que el programa no puede modificar su contenido. El lenguaje solamente permite las siguientes operaciones sobre variables de cadena:

- Asignación de referencias a cadenas (incluyendo nulo) a variables de cadena
- Concatenación de cadenas, usando el signo +
- Operadores de comparación, para comparar referencias de cadenas (no el contenido de la cadena).

Arreglos

El lenguaje permite que los programadores definan arreglos con elementos arbitarios de tipo: si T es un tipo, luego T[] es el tipo de arreglo con elementos del tipo T. Observe que la sintaxis para tipo de arreglo es recursiva, así que los elementos del arreglo pueden ser arreglos también. Los programadores pueden usar tal estructura para construir arreglos multidimensionales. Por ejemplo, el tipo T[][] describe un arreglo de dos dimensiones construido como un arreglo de arreglos referenciados del tipo T[].

Los arreglos son dinámicamente creados usando el constructor new: new T[n] localiza un arreglo del tipo T con n elementos e inicializa los elementos con sus valores por defecto. La expresión new T[n] evalua una referencia al arreglo recientemente creado.

Arreglos del tamaño n son indexados desde 0 a n – 1 y la notación normal de corchetes es usada para acceder elementos del arreglo. El lenguaje también provee un constructor para obtener la longitud del arreglo. Si la expresión a es una referencia a un arreglo de longitud n, entonces a.length es n, y a[i] retorna el i-esimo + 1 elemento del arreglo. Para cada acceso del arreglo a[i], el lenguaje provee chequeos en tiempo de ejecución para garantizar que a no sea nulo y que se accesa dentro de los límites: $0 \leq i < n$. Si el acceso al arreglo está fuera de sus límites, el programa termina con un mensaje de error.

Clases

Este lenguaje proporciona dos configuraciones estándar de lenguajes orientados a objetos: clases y herencia. Las clases son colecciones de campos y métodos. Ellas son definidas usando declaraciones de la forma: class A { body }, donde body es una secuencia de declaraciones de campos y métodos.

Las clases opcionalmente pueden extender otras clases existentes, usando el mecanismo normal de herencia en lenguajes orientados a objetos. Nuestro lenguaje permite que cada clase extienda a lo sumo otra clase, esto es, solamente soporta herencia simple. Las clases son heredadas usando la declaración de la forma: class A extends B { body }. Aquí, la clase A hereda todos los campos y métodos de la clase B; también se agrega los nuevos campos y métodos específicos en la declaración de A. Decimos que A es una subclase (o clase derivada) de B, y que B es una superclase (o clase base) de A. En general, definimos la relación de subclase como un cierre transitivo de la relación directa de

subclases descrita anteriormente. Es decir, si A es una subclase de B, y B es una subclase de C, entonces A es una subclase de C.

Requerimos que las clases extiendan solamente definiciones previas de clases. En otras palabras, una clase no puede extenderse a si misma o a otra clase definida mas tarde en el programa. Esto asegura que la jerarquía de clase tenga una estructura de arbol.

El programa no soporta sobrecarga de métodos. Una clase no puede tener multiples métodos con el mismo nombre, aún si los métodos tienen diferente número de tipos de argumentos, o retorna tipos diferentes. Requerimos que todos los métodos en cada una de las clases tengan diferentes nombres.

El lenguaje no permite ocultamiento de campos. Esto es, para cada declaración de clase class B extends A, todos los nuevos campos definidos de B deberán tener nombres diferentes a los campos heredados de A. De otra forma, estos campos heredados no podran ser accesados por los métodos de B.

Sin embargo, las subclases pueden redefinir (sobrescribir) métodos ya definidos por su superclase. En otras palabras, las subclases pueden definir versiones más especializadas de métodos existentes, los cuales accedan los campos adicionales y métodos de la subclase. Los métodos especializados son llamados métodos virtuales.

Subtipos

La herencia en clases via el constructor extends define una relación de subtipo entre sus tipos. Si la clase B extiende de A, luego B es un subtipo de A. Subtipos son siempre reflexivos (cada tipo de clase es un subtipo de si mismo) y es asimétrica (no hay tipos distintos los cuales son subtipos de otros). Aquí, la relación de subtipo es un orden parcial.

Subtipos garantizan la siguiente propiedad: Si B es un subtipo de A, entonces B tiene todas las características (es decir, campos y métodos) de A. Por lo tanto, un valor del tipo B puede ser usado en el programa donde un valor del tipo A es esperado para ser leido, por ejemplo en los argumentos de las funciones o en el lado derecho de una asignación.

Objetos

Objetos de una cierta clase pueden ser dinámicamente creados usando el constructor new. Si A es una clase declarada, new A() localiza un objeto de la clase A en el montículo e inicializa todos sus campos con el valor por defecto. La expresión new A() produce una referencia al nuevo objeto localizado.

Los campos y métodos de objeto son accesados usando '.': la expresión o.f denota el campo f del objeto o, y la expresión o.m() denota una llamada al método m del objeto o. La palabra reservada this referencia el objeto actual (es decir, el objeto sobre el cual el método actual es invocado).

Referencias de objetos tienen tipos de clases: cada definición class A introduce una clase de tipo A. Los tipos de clase pueden ser usados en declaraciones para referenciar variables. Por ejemplo, "A obj" declara una variable obj del tipo A, esto es, una referencia a un objeto de la clase A.

Un nombre de clase A puede ser usado como tipo de una referencia de objeto en cualquier parte del programa, con tal que el programa contenga (posiblemente letras) una declaración de class A. El lenguaje no requiere que la clase tipo A sea usada antes de la delaración de class A; en particular, el tipo A puede aparecer en el cuerpo de la declaración de la misma clase A, o incluso antes de (como una referencia posterior). Esto permite a los programadores construir recursiva y mutuamente estructuras recursivas de clases, tales como:

```
class List { int d; List next; }
class Node { int d; Edge[] edges; }
class Edge { Node to, from; }
```

Invocación de Métodos

Una invocación de un método consiste de los siguientes pasos: pasar los valores de los parámetros del llamante al llamado, ejecutar el cuerpo del llamado, y retornar el control y el valor resultante (si aplica) al llamante.

En cada lugar de invocación de un método, el programa **evalúa la expresión que representa los argumentos actuales** y luego asigna el valor calculado a los correspondientes parámetros formales del método. Argumentos objeto, arreglo o cadenas son pasados como referencia a tales estructuras. Los argumentos siempre son evaluados de izquierda a derecha.

Después de que se asigna los valores a los parámetros, el programa ejecuta el cuerpo del método invocado. Cuando la ejecución alcanza una instrucción return o alcanza el final del cuerpo del método, el programa transfiere el control al llamante. Si la instrucción return tiene una expresión como argumento, el argumento es evaluado y el valor calculado es también retornado al llamante.

A cada invocación del método, el número y tipo de los valores actuales en el lugar de la llamada debe ser el mismo al número y tipo de los parámetros formales en la declaración del método.

También, el tipo return de la declaración de un método debe coincidir con la instrucción return en el cuerpo del método. Más precisamente, si un método es declarado para retornar void, entonces la instrucción de retorno en el cuerpo del método no deberá tener una expresión return. En este caso, el método tiene permitido alcanzar el final de su cuerpo sin encontrar una instrucción return. Por otra parte, si el método es declarado con un return del tipo T, entonces la instrucción return debe retornar un valor del tipo T. En este caso, el cuerpo del método es permitido ejecutar una instrucción return antes de que alcance el final de su cuerpo.

Reglas de Ámbito

Para cada programa, hay una jerarquía de ámbitos consistentes de: el ámbito global, el ámbito de clases, el ámbito de método y el ámbito local para bloques dentro de los métodos. El ámbito global consiste del nombre de todas las clases definidas en el programa. El ámbito de una clase es el conjunto

de campos y métodos de la clase. El ámbito de las subclases son anidados dentro del ámbito de su superclase. El ámbito de un método consiste de los parámetros formales y variables locales definidos en el bloque representando el cuerpo del método. Finalmente, un ámbito de bloque contiene todas las variables definidas al comienzo del bloque. Al resolverse un identificador en cierto punto del programa, se búsca en lo ámbitos cercanos.

Existe un par de reglas para el ámbito. Primero, los identificadores pueden solamente ser usados si ellos son definidos en uno de los ámbitos cercanos. Más precisamente, las variables solo pueden ser usadas (leer o escribir) después de que ellas son definidas en uno de los bloques cercanos o ámbito de métodos. Campos y métodos pueden ser usados directamente (sin ningún prefijo) si el ámbito de la clase actual contiene estos campos o métodos, o ellos pueden ser usados en expresiones de la forma expr.f o expr.m cuando expr tiene tipo de clase T y el ámbito de T contiene estos campos o métodos.

Esto significa que todos los métodos y campos son públicos y pueden ser accesados por otras clases. Finalmente, los nombres de clase pueden ser usados en cualquier parte, con tal que ellos estén definidos en el programa (o antes o después del punto donde ellos se han referenciado).

Otra regla es que los identificadores (clases, campos, métodos y variables) no pueden ser definidos varias veces en el mismo ámbito. También, campos con el mismo nombre no pueden ser redefinidos en ámbitos de clases anidadas (es decir, en subclases). Por otra parte, identificadores pueden ser definidos varias veces en diferentes ámbitos, posiblemente anidados. Para las variables, los ámbitos internos ocultan los ámbitos externos. Esto es, si las variables con el mismo nombre ocurren en ámbitos anidados, entonces cada ocurrencia del nombre de la variable refiere a la variable en el ámbito más interno. Finalmente, no es permitido ocultar los parámetros de un método – las variables locales deben tener diferente nombre a los parámetros del método adjunto.

Declaraciones

Las declaraciones en este lenguaje son: asignaciones, llamadas a métodos, declaraciones de retorno, constructores **if**, ciclos **while**, declaraciones **break** y continue o declaraciones de bloque.

Cada declaración de asignación l = e actualiza la localización representada por l con el valor de la expresión e.

La localización actualizada l puede ser una variable local, un parámetro de un método, un campo de un objecto o un elemento de un arreglo.

El tipo de la localización actualizada debe coincidir con el tipo de la expresión evaluada. Para tipos enteros y lógicos, la asignación copia el valor entero o lógico. Para los tipos cadenas, arreglos u objetos, la asignación solamente copia la referencia al objeto (es decir, el objeto en sí no es copiado).

Las invocaciones a métodos pueden ser usadas como instrucciones, sin tener en cuenta si ellos devuelven valores o no. En el caso que el método invocado devuelve un valor, el valor es descartado.

La declaración if tiene la semántica normal. Consiste de una expresión de prueba condicional de tipo lógico, una rama para verdadero y una rama opcional para falso. Primero se evalúa la expresión de prueba. Si esta expresión evalúa un verdadero, se evalúa la rama de verdadero de la declaración. Si la expresión evalua un falso y la clausula else está presente, esta rama es ejecutada; de otra manera, el control es transferido a la siguiente instrucción después del if.

La clausula else siempre se refiere a la declaración del if más profundo.

La declaración while también tiene una semántica normal. Consiste de una expresión de prueba condicional de tipo lógico y un cuerpo del ciclo. La estructura del ciclo se ejecuta iterativamente. En cada iteración, se evalua la condición. Si la condición evalua un falso, entonce se finaliza la ejecución del ciclo; de otra manera se ejecuta el cuerpo del ciclo y continua con la proxima iteración.

La declaración de break y continue deben ocurrir en el cuerpo de un ciclo en el método actual.

La declaración break termina el ciclo y la ejecución continua en la siguiente declaración después del cuerpo del ciclo. La declaración continue termina la iteración actual del ciclo; la ejecución del programa procede a la siguiente iteración y prueba la condición del ciclo. Cuando una declaración break o continue ocurre en un ciclo anidado, se refiere al ciclo más interno.

Bloques de declaraciones consisten de una secuencia de declaraciones de variables locales, seguido por una secuencia de declaraciones.

Ya que los bloque son declaraciones en sí, significa que los bloques y declaraciones pueden ser anidados.

Expresiones

Una expresión puede ser una de las siguientes:

- El valor de una localización de memoria: variable local, parámetro de función, campo de objeto o elemento de arreglo.
- El valor retornado por llamada a un método. El método invocado debe tener declarado un tipo T diferente a void.
- El objeto actual this. El tipo de this es el de la clase adjunta.
- Una constructor new, la cual crea un objeto o un arreglo. En la expresión new T(), tipo T debe ser un tipo objeto. En la expresión new T[exp.], expr debe tener un tipo entero y T puede tener cualquier tipo.
- La expresión expr.length representa la longitud de un arreglo. Aquí, expr debe tener tipo de arreglo.
- Una expresión unaria o binaria construida con los operadores presentados abajo.
- Una literal, como la descrita en la sección acerca de consideraciones lexicas.
- Expresión encerrada en paréntesis, para cambiar o hacer explícita la precedencia de operadores.

Reglas de tipo para Expresiones

$$\frac{}{A \vdash \text{true} : \text{boolean}}$$

$$\frac{}{A \vdash \text{false} : \text{boolean}}$$

$$\frac{}{A \vdash \textit{integer-literal} : \text{int}}$$

$$\frac{}{A \vdash \textit{string-literal} : \text{string}}$$

$$\frac{A \vdash E_0 : \text{int} \quad A \vdash E_1 : \text{int}}{A \vdash E_0 \ op \ E_1 : \text{int}} \quad op \in \{\texttt{+}, \texttt{-}, \texttt{/}, \texttt{*}, \texttt{\%}\}$$

$$\frac{A \vdash E_0 : \text{string} \quad A \vdash E_1 : \text{string}}{A \vdash E_0 \ + \ E_1 : \text{string}}$$

$$\frac{A \vdash E_0 : T \quad A \vdash E_1 : T}{A \vdash E_0 \ op \ E_1 : \text{boolean}} \quad op \in \{\texttt{==}, \texttt{!=}\}$$

$$\frac{A \vdash E_0 : \text{int} \quad A \vdash E_1 : \text{int}}{A \vdash E_0 \ op \ E_1 : \text{boolean}} \quad op \in \{\texttt{<=}, \texttt{<}, \texttt{>=}, \texttt{>}\}$$

$$\frac{A \vdash E_0 : \text{boolean} \quad A \vdash E_1 : \text{boolean}}{A \vdash E_0 \ op \ E_1 : \text{boolean}} \quad op \in \{\texttt{\&\&}, \texttt{||}\}$$

$$\frac{A \vdash E : \text{int}}{A \vdash \texttt{-}E : \text{int}}$$

$$\frac{A \vdash E : \text{boolean}}{A \vdash \texttt{!}E : \text{boolean}}$$

$$\frac{A \vdash E_0 : T\texttt{[]} \quad A \vdash E_1 : \text{int}}{A \vdash E_0[E_1] : T}$$

$$\frac{A \vdash E : \text{int}}{A \vdash \text{new } T[E] : T\texttt{[]}}$$

$$\frac{A \vdash E : T\texttt{[]}}{A \vdash E\texttt{.}\,\text{length} : \text{int}}$$

$$\frac{A \vdash E_0 : T_1 \times \ldots \times T_n \to T \quad A \vdash E_i : T_i,\ 1 \le i \le n}{A \vdash E_0(E_1, \ldots, E_n) : T}$$

$$\frac{T \in C}{A \vdash \text{new } T\texttt{()} : T}$$

$$\frac{A \vdash E : T \quad T \in C \quad (id : T') \in T}{A \vdash E\texttt{.}\,id : T'}$$

$$\frac{id : T \in A}{A \vdash id : T}$$

Reglas de Tipo para Instrucciones

$$\frac{A \vdash E : T' \quad T' \leq T \quad A, \mathsf{x} : T \vdash S : \mathsf{unit}}{A \vdash T\ \mathsf{x} = E\ ;\ S : \mathsf{unit}}\ (\textsc{Decl1}) \qquad \frac{A, \mathsf{x} : T \vdash S : \mathsf{unit}}{A \vdash T\ \mathsf{x}\ ;\ S : \mathsf{unit}}\ (\textsc{Decl2})$$

$$\frac{A \vdash S_1 : \mathsf{unit} \quad S_1 \text{ not declaration} \quad A \vdash S_2 : \mathsf{unit}}{A \vdash S_1\ ;\ S_2 : \mathsf{unit}}\ (\textsc{Seq}) \qquad \frac{A \vdash L : T \quad A \vdash E : T' \quad T' \leq T}{A \vdash L = E : \mathsf{unit}}\ (\textsc{Assign})$$

$$\frac{A \vdash E : \mathsf{boolean} \quad A \vdash S_1 : \mathsf{unit} \quad A \vdash S_2 : \mathsf{unit}}{A \vdash \mathsf{if}\ (E)\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2 : \mathsf{unit}}\ (\textsc{If}) \qquad \frac{A \vdash E : \mathsf{boolean} \quad A \vdash S : \mathsf{unit}}{A \vdash \mathsf{while}\ (E)\ S : \mathsf{unit}}\ (\textsc{While})$$

Código para Quicksort

La siguiente clase muestra un ejemplo de código de MJ el cual ordena un arreglo de enteros.

```
class Test {
      int partition(int [] a, int low, int high) {
        int pivot = a[low];
        int i = low;
        int j = high;
        int tmp;

        while (true) {
           while (a[i] < pivot) i = i+1;
           while (a[j] > pivot) j = j-1;
           if (i >= j) break;
           tmp = a[i];
           a[i] = a[j];
           a[j] = tmp;
           i = i + 1;
           j = j - 1;
        }
        return j;
    }

    void quicksort(int [] a, int low, int high) {
        if (low < high) {
           int mid = partition(a, low, high);
           quicksort(a, low, mid);
           quicksort(a, mid+1, high);
        }
    }
}
```

```
  program ::= classDecl*
classDecl ::= class id [extends id] '{' (fieldDecl | methDecl)* '}'

fieldDecl ::= type id ( ',' id )* ';'
 methDecl ::= (type | void) id '(' [formals] ')' block
  formals ::= type id ( ',' type id)*

    type ::= int | boolean | string | id | type '[' ']'

   block ::= '{' varDecl* stmt* '}'
 varDecl ::= type id ['=' expr] ( ',' id ['=' expr] )* ';'

    stmt ::= assign ';'
           | call ';'
           | return [expr] ';'
           | if '(' expr ')' stmt [else stmt]
           | while '(' expr ')' stmt
           | break ';' | continue ';'
           | block

  assign ::= location '=' expr
location ::= id | expr '.' id | expr '[' expr ']'

    call ::= method '(' [actuals] ')'
  method ::= id | expr '.' id
 actuals ::= expr (',' expr)*

    expr ::= location
           | call
           | this
           | new id '(' ')'
           | new type '[' expr ']'
           | expr '.' length
           | expr binary expr
           | unary expr
           | literal
           | '(' expr ')'

  binary ::= '+' | '-' | '*' | '/' | '%' | '&&' | '||'
           |  '<' | '<=' | '>' | '>=' | '==' | '!='
   unary ::= '-' | '!'
 literal ::= integer-literal | string-literal | true | false | null
```

Figura 1: Gramática del lenguaje MJ