Analizador Léxico
Compiladores e Intérpretes
2012

Garay, Iñaki LU 67387

28 de agosto de 2012

Índice general

0.1. Uso

0.1.1. Invocación

Utilizando el intérprete de Python para correr el script del módulo principal:

```
C:\> python.exe main.py <archivo_entrada> [<archivo_salida>]
```

La especificación del archivo de salida es opcional. Si se omite, el analizador léxico mostrará su salida por pantalla. Si no se especifica un archivo de entrada, o se pasan más argumentos que el archivo de entrada y el de salida, el analizador léxico mostrará un mensaje explicando el uso por linea de comando.

0.1.2. Interpretación de Salida

Mientras procesa el archivo, el analizador léxico mostrará en su salida los tokens reconocidos en una tabla, con el siguiente formato:

```
$ python2 main.py ./tests/006-err-ampersand.java
1:0     -   <INT_TYPE>      :: int
1:3     -   <SEPARATOR>     ::
1:4     -   <IDENTIFIER>    :: fun
1:7     -   <PAREN_OPEN>    :: (
1:8     -   <INT_TYPE>      :: int
1:11    -   <SEPARATOR>     ::
1:12    -   <IDENTIFIER>    :: a
1:13    -   <PAREN_CLOSE>   :: )
1:14    -   <SEPARATOR>     ::
1:15    -   <BRACE_OPEN>    :: {
1:16    -   <SEPARATOR>     ::
2:4     -   <IDENTIFIER>    :: un
2:6     -   <SEPARATOR>     ::
ERROR: Line: 2, Col: 7 :: Token no reconocido.
In line 2:7
    un & amper;
-------^
```

La primera columna de la tabla indica el número de linea y la columna en la cual se encontró el token. La siguiente columna indica el tipo del token. La tercera columna muestra el lexema del token.

0.1.3. Bugs conocidos

- El reporte de la columna en la cual se detecta un error léxico no siempre es correcto.
- El reporte de la linea en la cual se detecta un error léxico no es correcto si la linea anterior termina en espacios.
- La detección del operador prohibido `<<<=` no es correcto.

0.2. Análisis Léxico

0.2.1. Alfabeto de entrada

El alfabeto de entrada son todos los caracteres de la codificación ASCII.

0.2.2. Definición de Tokens

Las expresiones regulares se expresaron utilizando la sintaxis válida para JLex.

La siguiente tabla muestra los tokens reconocidos por el analizador léxico.

Token	**Expresión Regular**	**Ejemplo**
TK_IDENTIFIER	[a-zA-Z_\$][a-zA-Z_\$0-9]*	identifier
TK_INT_LITERAL	(0|[1-9]([0-9])*)	42
TK_CHAR_LITERAL	(\'[^(’\\)]\’)|(\’[\\\\\|\\’|\\\"|\\n]\’)	'c'
TK_STRING_LITERAL	(\"\"|\"([^(\"\\)]|[\\\\\|\\’|\\\"|\\n])*\")	"s"
TK_BRACE_OPEN	{	{
TK_BRACE_CLOSE	}	}
TK_PAREN_OPEN	\)	(
TK_PAREN_CLOSE	\(	)
TK_PERIOD	.	.
TK_COMMA	,	,
TK_SEMICOLON	;	;
TK_ASSIGNMENT	=	=
TK_ADD	+	+
TK_SUB	-	-
TK_MUL	*	*
TK_DIV	\/	/
TK_MOD	%	%
TK_LT	<	<
TK_GT	>	>
TK_LTEQ	<=	< =
TK_GTEQ	>=	> =
TK_EQUALS	==	==
TK_NOTEQUALS	!=	!=
TK_NOT	!	!
TK_AND	&&	&&
TK_OR	||	||
TK_BOOLEAN	boolean	boolean
TK_CHAR	char	char
TK_CLASS	class	class
TK_CLASSDEF	classDef	classDef
TK_ELSE	else	else
TK_EXTENDS	extends	extends
TK_FALSE	false	false
TK_FOR	for	for
TK_IF	if	if
TK_INT	int	int

Token	**Expresión Regular**	**Ejemplo**
TK_NEW	new	new
TK_NULL	null	null
TK_RETURN	return	return
TK_STRING	String	String
TK_SUPER	super	super
TK_THIS	this	this
TK_TRUE	true	true
TK_VOID	void	void
TK_WHILE	while	while
TK_EOF	<EOF>	<EOF>

0.2.3. Operadores Prohibidos

Operador	Expresión Regular
~	~
?	?
:	:
++	++
–	–
&	&
|	|
^	^
<<	<<
>>	>>
>>>	>>>
+=	+=
-=	-=
=	*=
/=	/=
&=	&=
|=	|=
^=	^=
%=	%=
<<=	<<=
>>=	>>=
>>>=	>>>=

0.2.4. Palabras Prohibidas

```
abstract        interface
break
byte            long
                native
byvalue         none
case            operator
cast            outer
catch           package
```

```
const          private
continue       protected
default        public
do             rest
double
final          short
finally        static
float          switch
future         synchronized
generic        throw
goto           throws
implements     transient
import         try
inner          var
instanceof     volatile
```

0.3. Gramática

0.3.1. Gramatica original

```
<start>                 ::= <class>+

<class>                 ::= class <identifier> <super>?
                            { <field>* <ctor>* <method>* }

<super>                 ::= extends <identifier>

<field>                 ::= <type> <var_declarator_list> ;

<method>                ::= <method_type> <identifier>
                            <formal_args> <block>

<ctor>                  ::= <identifier> <formal_args> <block>

<formal_args>           ::= ( <formal_arg_list>? )

<formal_arg_list>       ::= <formal_arg>
<formal_arg_list>       ::= <formal_arg> , <formal_arg_list>

<formal_arg>            ::= <type> <identifier>

<method_type>           ::= void
<method_type>           ::= <type>

<type>                  ::= <primitive_type>
<type>                  ::= <identifier>

<primitive_type>        ::= boolean
```

```
<primitive_type>        ::= char
<primitive_type>        ::= int
<primitive_type>        ::= String

<var_declarator_list>   ::= <identifier> , <var_declarator_list>
<var_declarator_list>   ::= <identifier>

<block>                 ::= { <statement>* }

<statement>             ::= ;
<statement>             ::= if ( <expression> ) <statement>
<statement>             ::= if ( <expression> ) <statement>
                            else <statement>
<statement>             ::= return <expression>? ;
<statement>             ::= <block>

<expressions>           ::= <expression> <binary_op> <expression>
<expressions>           ::= <unary_op> <expression>
<expressions>           ::= <primary>

<binary_op>             ::= =
<binary_op>             ::= ||
<binary_op>             ::= &&
<binary_op>             ::= ==
<binary_op>             ::= !=
<binary_op>             ::= <
<binary_op>             ::= >
<binary_op>             ::= <=
<binary_op>             ::= >=
<binary_op>             ::= +
<binary_op>             ::= -
<binary_op>             ::= *
<binary_op>             ::= /
<binary_op>             ::= %

<unary_op>              ::= !
<unary_op>              ::= +
<unary_op>              ::= -

<primary>               ::= <new_expr>
<primery>               ::= <identifier>

<new_expr>              ::= <literal>
<new_expr>              ::= this
<new_expr>              ::= this . <identifier>
<new_expr>              ::= ( <expression> )
<new_expr>              ::= new <identifier>
                            <actual_args>
<new_expr>              ::= <identifier> <actual_args>
<new_expr>              ::= <primary> . <identifier>
```

```
                            <actual_args>
<new_expr>                ::= super . <identifier>
                            <actual_args>

<literal>                 ::= null
<literal>                 ::= true
<literal>                 ::= false
<literal>                 ::= <int_literal>
<literal>                 ::= <char_literal>
<literal>                 ::= <string_literal>

<actual_args>             ::= ( <expr_list>? )

<expr_list>               ::= <expression>
<expr_list>               ::= <expression> , <expr_list>
```

0.3.2. Gramatica modificada

Cambios realizados:

- Producciones para `classDef`
- Producciones para expresiones, tomando en cuenta precedencia y asociatividad.
- Producciones para sentencia `while`.
- Producciones para sentencia `for`.

```
<start>                   ::= <classdef>+ <class>+

<classdef>                ::= classdef <identifier> <super>?
                              { <field>* <classdef_ctor>*
                              <classdef_method>* }

<class>                   ::= class <identifier> <super>?
                              { <field>* <ctor>* <method>* }

<super>                   ::= extends <identifier>

<field>                   ::= <type> <var_declarator_list> ;

<classdef_method>         ::= <method_type> <identifier>
                              <formal_args>

<method>                  ::= <method_type> <identifier>
                              <formal_args> <block>

<classdef_ctor>           ::= <identifier> <formal_args>
```

```
<ctor>                  ::= <identifier> <formal_args> <block>

<formal_args>           ::= ( <formal_arg_list>? )

<formal_arg_list>       ::= <formal_arg>
<formal_arg_list>       ::= <formal_arg> , <formal_arg_list>

<formal_arg>            ::= <type> <identifier>

<method_type>           ::= void
<method_type>           ::= <type>

<type>                  ::= <primitive_type>
<type>                  ::= <identifier>

<primitive_type>        ::= boolean
<primitive_type>        ::= char
<primitive_type>        ::= int
<primitive_type>        ::= String

<var_declarator_list>   ::= <identifier> , <var_declarator_list>
<var_declarator_list>   ::= <identifier>

<block>                 ::= { <statement>* }

<statement>             ::= ;
<statement>             ::= if ( <expression> ) <statement>
<statement>             ::= if ( <expression> ) <statement>
                            else <statement>
<statement>             ::= while ( <statement> ) <statement>
<statement>             ::= for ( <statement> ; <statement> ;
                            <statement> ) <statement>
<statement>             ::= return <expression>? ;
<statement>             ::= <block>

<expression>            ::= <assignment_expr>

<assignment_expr>       ::= <logical_expr>

<logical_expr>          ::= <logical_or_expr> <logical_expr_rest>

<logical_expr_rest>     ::= LAMBDA
<logical_expr_rest>     ::= ASSIGNMENT <logical_expr>

<logical_or_expr>       ::= <logical_and_expr> <logical_or_expr_rest>

<logical_or_expr_rest>  ::= LAMBDA
<logical_or_expr_rest>  ::= OR <logical_or_expr>
```

```
<logical_and_expr>      ::= <equality_expr> <logical_and_expr_rest>

<logical_and_expr_rest> ::= LAMBDA
<logical_and_expr_rest> ::= AND <logical_and_expr>

<equality_expr>         ::= <relational_expr> <equality_expr_rest>

<equality_expr_rest>    ::= LAMBDA
<equality_expr_rest>    ::= EQUALS <equality_expr>
<equality_expr_rest>    ::= NOTEQUALS <equality_expr>

<relational_expr>       ::= <term_expr> <relational_expr_rest>

<relational_expr_rest>  ::= LAMBDA
<relational_expr_rest>  ::= LT <relational_expr>
<relational_expr_rest>  ::= GT <relational_expr>
<relational_expr_rest>  ::= LTEQ <relational_expr>
<relational_expr_rest>  ::= GTEQ <relational_expr>

<term_expr>             ::= <factor_expr> <additive_expr_rest>

<additive_expr_rest>    ::= LAMBDA
<additive_expr_rest>    ::= ADD <term_expr>
<additive_expr_rest>    ::= SUB <term_expr>

<factor_expr>           ::= <unary_expr> <multiplicative_expr_rest>

<factor_expr_rest>      ::= LAMBDA
<factor_expr_rest>      ::= MUL <factor_expr>
<factor_expr_rest>      ::= DIV <factor_expr>
<factor_expr_rest>      ::= MOD <factor_expr>

<unary_expr>            ::= ADD <unary_expr>
<unary_expr>            ::= SUB <unary_expr>
<unary_expr>            ::= NOT <unary_expr>
<unary_expr>            ::= <primary>

<primary>               ::= <new_expr>
<primery>               ::= <identifier>

<new_expr>              ::= <literal>
<new_expr>              ::= this
<new_expr>              ::= this . <identifier>
<new_expr>              ::= ( <expression> )
<new_expr>              ::= new <identifier> <actual_args>
<new_expr>              ::= <identifier> <actual_args>
<new_expr>              ::= <primary> . <identifier> <actual_args>
<new_expr>              ::= super . <identifier> <actual_args>

<literal>               ::= null
```

```
<literal>               ::= true
<literal>               ::= false
<literal>               ::= <int_literal>
<literal>               ::= <char_literal>
<literal>               ::= <string_literal\textless{}

<actual_args>           ::= ( \textless{}expr_list>? )

<expr_list>             ::= <expression\textless{}
<expr_list>             ::= <expression> , <expr_list>
```

<subsectionGramatica modificada 2

La siguiente gramatica es identica a la anterior con los siguientes cambios:

- Los simbolos terminales has sido reemplazados por los tokens producidos por el scanner.
- Se eliminaron las extensiones en la gramatica, eliminando el uso de los +, *, ?.
- Se agrego una produccion de <start> para eliminar el +.
- Se agrego una produccion de <class> para eliminar el <super>?
- Se agrego una produccion <class_body> para factorizar field ctor method .
- Se agregaron las producciones <fields> <ctors> y <methods> para eliminar el <field>*.
- Se agrego un simbolo <statements> para el <statement>* en la produccion de <block>.
- Se agrego una produccion de <statement> para el return <expression>? ;
- agregada produccion <actual_args> para <expr_list>?.

```
<start>                 ::= <class>
<start>                 ::= <class> <start>

<class>                 ::= TK_CLASS TK_IDENTIFIER <class_body>
<class>                 ::= TK_CLASS TK_IDENTIFIER <super> <class_body>

<class_body>            ::= TK_BRACE_OPEN
                            <fields> <ctors> <methods>
                            TK_BRACE_CLOSE

<fields>                ::= LAMBDA
<fields>                ::= <field>
<fields>                ::= <field> <fields>
```

```
<ctors>                 ::= LAMBDA
<ctors>                 ::= <ctor>
<ctors>                 ::= <ctor> <ctors>

<methods>               ::= LAMBDA
<methods>               ::= <method>
<methods>               ::= <method> <methods>

<super>                 ::= TK_EXTENDS TK_IDENTIFIER

<field>                 ::= <type> <var_declarator_list> TK_SEMICOLON

<method>                ::= <method_type> TK_IDENTIFIER
                            <formal_args> <block>

<ctor>                  ::= TK_IDENTIFIER <formal_args> <block>

<formal_args>           ::= TK_PAREN_OPEN TK_PAREN_CLOSE
<formal_args>           ::= TK_PAREN_OPEN
                            <formal_arg_list>
                            TK_PAREN_CLOSE

<formal_arg_list>       ::= <formal_arg>
<formal_arg_list>       ::= <formal_arg> TK_COMMA <formal_arg_list>

<formal_arg>            ::= <type> TK_IDENTIFIER

<method_type>           ::= TK_VOID
<method_type>           ::= <type>

<type>                  ::= TK_IDENTIFIER
<type>                  ::= <primitive_type>

<primitive_type>        ::= TK_BOOLEAN
<primitive_type>        ::= TK_CHAR
<primitive_type>        ::= TK_INT
<primitive_type>        ::= TK_STRING

<var_declarator_list>   ::= TK_IDENTIFIER
<var_declarator_list>   ::= TK_IDENTIFIER TK_COMMA
                            <var_declarator_list>

<block>                 ::= TK_BRACE_OPEN <statements> TK_BRACE_CLOSE

<statements>            ::= <statement>
<statements>            ::= <statement> <statements>

<statement>             ::= TK_SEMICOLON
<statement>             ::= TK_IF TK_PAREN_OPEN <expression>
                            TK_PAREN_CLOSE <statement>
```

```
<statement>             ::= TK_IF
                            TK_PAREN_OPEN <expression>
                            TK_PAREN_CLOSE <statement>
                            TK_ELSE <statement>
<statement>             ::= TK_RETURN TK_SEMICOLON
<statement>             ::= TK_RETURN <expression> TK_SEMICOLON
<statement>             ::= <block>
<statement>             ::= TK_FOR TK_PAREN_OPEN <statement>
                            TK_SEMICOLON <statement> TK_SEMICOLON
                            <statement> TK_PAREN_CLOSE <statement>
<statement>             ::= TK_WHILE TK_PAREN_OPEN <statement>
                            TK_PAREN_CLOSE <statement>

<expression>            ::= <assignment_expr>

<assignment_expr>       ::= <logical_expr>

<logical_expr>          ::= <logical_or_expr> <logical_expr_rest>

<logical_expr_rest>     ::= LAMBDA
<logical_expr_rest>     ::= TK_ASSIGNMENT <logical_expr>

<logical_or_expr>       ::= <logical_and_expr> <logical_or_expr_rest>

<logical_or_expr_rest>  ::= LAMBDA
<logical_or_expr_rest>  ::= TK_OR <logical_or_expr>

<logical_and_expr>      ::= <equality_expr> <logical_and_expr_rest>

<logical_and_expr_rest> ::= LAMBDA
<logical_and_expr_rest> ::= TK_AND <logical_and_expr>

<equality_expr>         ::= <relational_expr> <equality_expr_rest>

<equality_expr_rest>    ::= LAMBDA
<equality_expr_rest>    ::= TK_EQUALS <equality_expr>
<equality_expr_rest>    ::= TK_NOTEQUALS <equality_expr>

<relational_expr>       ::= <term_expr> <relational_expr_rest>

<relational_expr_rest>  ::= LAMBDA
<relational_expr_rest>  ::= TK_LT <relational_expr>
<relational_expr_rest>  ::= TK_GT <relational_expr>
<relational_expr_rest>  ::= TK_LTEQ <relational_expr>
<relational_expr_rest>  ::= TK_GTEQ <relational_expr>

<term_expr>             ::= <factor_expr> <additive_expr_rest>

<additive_expr_rest>    ::= LAMBDA
<additive_expr_rest>    ::= TK_ADD <term_expr>
```

```
<additive_expr_rest>    ::= TK_SUB <term_expr>

<factor_expr>           ::= <unary_expr> <multiplicative_expr_rest>

<factor_expr_rest>      ::= LAMBDA
<factor_expr_rest>      ::= TK_MUL <factor_expr>
<factor_expr_rest>      ::= TK_DIV <factor_expr>
<factor_expr_rest>      ::= TK_MOD <factor_expr>

<unary_expr>            ::= TK_ADD <unary_expr>
<unary_expr>            ::= TK_SUB <unary_expr>
<unary_expr>            ::= TK_NOT <unary_expr>
<unary_expr>            ::= <primary>

<primary>               ::= TK_IDENTIFIER
<primary>               ::= <new_expr>

<new_expr>              ::= <literal>
<new_expr>              ::= TK_THIS
<new_expr>              ::= TK_THIS TK_PERIOD TK_IDENTIFIER
<new_expr>              ::= TK_PAREN_OPEN <expression> TK_PAREN_CLOSE
<new_expr>              ::= TK_NEW TK_IDENTIFIER <actual_args>
<new_expr>              ::= TK_IDENTIFIER <actual_args>
<new_expr>              ::= <primary> TK_PERIOD
                            TK_IDENTIFIER <actual_args>
<new_expr>              ::= TK_SUPER TK_PERIOD
                            TK_IDENTIFIER <actual_args>

<literal>               ::= TK_NULL
<literal>               ::= TK_TRUE
<literal>               ::= TK_FALSE
<literal>               ::= TK_INT_LITERAL
<literal>               ::= TK_CHAR_LITERAL
<literal>               ::= TK_STRING_LITERAL

<actual_args>           ::= TK_PAREN_OPEN TK_PAREN_CLOSE
<actual_args>           ::= TK_PAREN_OPEN <expr_list> TK_PAREN_CLOSE

<expr_list>             ::= <expression>
<expr_list>             ::= <expression> TK_COMMA <expr_list>
```

0.4. Diseño

La principal decisión de diseño que afecto la implementación fue la abstracción de los estados del automata finito reconocedor.

El analizador léxico se desarrolló utilizando únicamente la versión 2.7 del lenguaje Python (+www.python.org+).

Para implementar el analizador léxico se realizó la especificación de la má-

quina de estados basándose en las expresiones regulares definidas en la sección anterior, para luego representarlo en código Python.

0.4.1. Archivo `main.py`

Contiene el punto de inicio de ejecución del programa, instanciando el `Scanner` con el archivo de entrada, y realiza checkeo de errores en la invocacíón.

0.4.2. Archivo `scanner.py`

Contiene la definición de la clase `Scanner`.

0.4.3. Archivo `states.py`

Contiene la definicion de la clase `State`, la instanciación de los estados del autómata finito reconocedor y sus transiciones.

0.4.4. Archivo `tokens.py`

Contiene la definición de la clase `Token` y `TokenType`, y la instanciación de los tokens retornados por el analizador léxico.

0.4.5. Archivo `errors.py`

Contiene la definición de la clase `LexicalError`.

0.4.6. Clase `Scanner`

`Scanner` es la clase que representa al analizador léxico propiamente. Este implementa el método `get_token()` que devuelve secuencialmente todos los tokens reconocidos en un dado archivo que contiene código `MiniJava-Decaf`.

0.4.7. Clase `State`

La clase principal utilizada es la llamada `State`. Esta representa una abstracción de un estado de un autómata finito, que cuenta con una serie de funciones de checkeo para determinar si se activa una transición o no.

0.4.8. Clases `Token` y `TokenType`

Para representar a los tokens se armaron dos clases: `TokenType` y `Token`. `TokenType`, definida en `tokens.py`, es una abstracción sobre los distintos tipos de tokens reconocidos y especificados en la sección anterior. `Token` abstrae un token y almacena los metadatos asociados a el, y es instanciado según el análisis del archivo de código fuente en cuestión.

0.4.9. Clase `LexicalError`

Para el manejo de errores, se creó el tipo de excepción `LexicalError` dentro del archivo `errors.py`.

0.5. Verificación

0.5.1. Errores detectados

El analizador léxico reconoce los siguientes tipos de errores:

- Comentario del tipo /* */ que no esté propiamente cerrado.
- Informa de tokens no reconocidos.
- Caracter no reconocido: si se intenta ingresar un caracter que no pertenece al alfabeto se producirá un error.
- Si el archivo de entrada especificado no existe, se producirá un error.
- Si se encuentra un caracter literal de mas de un caracter, e.g. 'hola', se producirá un error. En cambio, si es un caracter válido, e.g. 'n', se aceptará.

0.5.2. Casos de prueba

Cuando un error es detectado, también se muestra por pantalla la ubicación de la porción del texto que presenta el error. Por ejemplo:

```
ERROR: Line: 3, Col: 4 :: Comentario no cerrado.
In line 3:4
    /* comment
----^

ERROR: Line: 2, Col: 10 :: Token no reconocido.
In line 2:10
    a = 1 # 2
----------^
```

Test `000-cor-tokens.java`

Descripción: El archivo contiene todos los tokens reconocidos por el scanner, uno por linea.

Resultado esperado: Caso correcto.

Test `001-err-forbidden-words.java`

Descripción: El archivo contiene todas las palabras prohibidas, una por linea, comentadas. El analizador léxico falla al detectar la primera, se puede modificar el archivo descomentando las palabras para probar la detección de cada una de las palabras.

Resultado esperado: Caso erróneo. detección correcta de palabras prohibidas, emitiendo un error y un mensaje apropiado.

Test `002-err-forbidden-operator.java`

Descripción: El archivo contiene todos los operadores prohibidos, uno por linea, comentados. El analizador léxico falla al detectar el primero; se puede modificar el archivo descomentando los otros operadores para probar la detección de cada uno.

Resultado esperado: Caso erróneo.

Test `003-cor-ejemplo.java`

Descripción: Este es el caso de test principal para detección correcta de tokens.

Resultado esperado: Caso correcto. Todos los tokens son reconocidos con éxito según lo esperado, y los comentarios son obviados sin modificar la posición final de los tokens alrededor de ellos.

Test `004-err-hash.java`

Descripción: El archivo contiene un ejemplo de un operador erróneo en el contexto de una función.

Resultado esperado: Caso erróneo.

Test `005-err-invalid-token.java`

Descripción: El archivo contiene un ejemplo de un token inválido.

Resultado esperado: Caso erróneo.

Test `006-err-ampersand.java`

Descripción: El archivo contiene un ejemplo de un token inválido.

Resultado esperado: Caso erróneo.

Test `007-err-comentario no finalizado.java`

Descripción: El archivo contiene un ejemplo de un comentario no cerrado.

Resultado esperado: Caso erróneo.

Test `008-cor-comments.java`

Descripción: El archivo contiene código con varios ejemplos de comentarios. La idea de este test es mostrar que aun con comentarios, los números de linea y columna siguen siendo calculados correctamente.

Resultado esperado: Caso correcto.

Test `009-err-invalid-lit-char.java`

Descripción: El archivo contiene un ejemplo con un literal de caractér inválido.

Resultado esperado: Caso erróneo.

Test `010-err-string-no-cerrado.java`

Descripción: El archivo contiene un ejemplo con un string no cerrado.

Resultado esperado: Caso erróneo.

Test `011-err-string-con-newline.java`

Descripción: El archivo contiene un ejemplo con un string que no esta cerrado en la misma linea.

Resultado esperado: Caso erróneo.

Test `012-intlit-malformado.java`

Descripción: El archivo contiene un ejemplo con un literal de entero mal formado.

Resultado esperado: Caso erróneo.

Test `113-strlit-malformado.java`

Descripción: El archivo contiene un ejemplo con un literal de string mal formado.

Resultado esperado: Caso erróneo.

Test `114-charlit-malformado.java`

Descripción: El archivo contiene un ejemplo con un litearl de caractér mal formado.

Resultado esperado: Caso erróneo.