Proyecto Compiladores e Intérpretes

MINIJAVA

Matias Emiliano Gandolfo
LU: 122002

Analizador léxico

Etapa 1

Compilación de código fuente

Para compilar el código fuente se debe seguir el estándar, y una vez se obtenga el .jar, la ejecución del analizador léxico es equivalente a la presentada en clase. Utilizando el siguiente comando el analizador léxico funcionará correctamente dado un archivo.

```
java -jar Compilador.jar ruta/nombre_archivo_analizar.extensión
```

Token reconocidos por el analizador léxico

Sea X un conjunto de caracteres, el operador ^X indica que el caracter que va en esa posición, es cualquiera que no pertenezca al conjunto X.

Conjunto universal Σ representa a todos los caracteres del lenguaje menos EOF.

El subconjunto de Σ, Σaux representa todos los caracteres de Σ menos '\n'.

Token entero

Un literal entero es una secuencia de uno o más dígitos. El literal entero no puede contener más de 9 dígitos y su valor corresponde a su interpretación estándar en base 10. Para esta cadena se utiliza el token **literalInteger**.

Expresión regular → **[0..9]{1,9}**

Token comentario única línea

Expresión regular → **/ / Σaux**

Token comentario multilínea

Expresión regular → **/ * Σ * /**

Token identificador

Un identificador de clase es una letra mayúscula seguida de cero o más letras (mayúsculas o minúsculas), dígitos y underscores. Para esta cadena se utiliza el token **idClass**.

Un identificador de método o variable es una letra minúscula seguida de cero o más letras (mayús. o minúsculas), dígitos y underscores. Para esta cadena se utiliza el token **idMetVar**.

Expresión regular → **([a..z] | [A..Z]) ([a..z] | [A..Z] | [0..9] | _) ***

Antes de aceptar una cadena válida y devolverla como un identificador, se chequea que esta no sea una palabra reservada utilizando el mapeo que almacena la clase ReservedWords. En caso de que el identificador se corresponda con una palabra reservada el token será **idKeyWord**.

Token caracter

Puede tener cualquiera de las siguiente formas:

- 'x' donde x es cualquier caracter excepto la barra invertida (\) o la comilla simple (').
- '\x' donde x es cualquier caracter.
- '\uxxxx' donde x es un dígito en hexadecimal con un rango entre 0000 y FFFF (LOGRO).

Para reconocer estas cadenas se utiliza el token **literalCharacter**.

Expresión regular 'x' → **' [^ (\ | ')] '**
Expresión regular '\x' → **' \ [Σaux] '**
Expresión regular '\u0000' → **' \ u ([0..9] | [A...F]){4} '**

Token string

Un literal string se representa mediante una comilla doble (") seguida de una secuencia de caracteres y finaliza con otra comilla doble ("). La secuencia de caracteres no puede ser interrumpida por un salto de línea. Es posible que en la secuencia de caracteres aparezca una comilla doble (") siempre y cuando sea antecedida por una barra invertida (\). Para esta cadena se utiliza el token **literalSring**.

Expresión regular → **" (^ (\ ))* | (\ Σ)* "**

Token puntuación

Este tipo de tokens no posee una sintaxis, son caracteres sueltos individuales.

Expresión regular (→ **(** Token: **punctuationOpeningParenthesis**.
Expresión regular) → **)** Token: **punctuationClosingParenthesis**.
Expresión regular { → **{** Token: **punctuationOpeningBracket**.
Expresión regular } → **}** Token: **punctuationClosingBracket**.
Expresión regular ; → **;** Token: **punctuationSemicolon**.
Expresión regular , → **,** Token: **punctuationComma**.
Expresión regular . → **.** Token: **punctuationPoint**.

Token operador

Estos tokens pueden formar operadores simples y algunos otros compuestos.

Expresión regular > → **>** Token: **opGreater**.
Expresión regular < → **<** Token: **opLess**.
Expresión regular ! → **!** Token: **opNegation**.
Expresión regular == → **= =** Token: **opEqual**.
Expresión regular >= → **> =** Token: **opGreaterOrEqual**.
Expresión regular <= → **< =** Token: **opLessOrEqual**.
Expresión regular != → **! =** Token: **opDistinct**.
Expresión regular + → **+** Token: **opAddition**.
Expresión regular - → **-** Token: **opSubtraction**.
Expresión regular * → ***** Token: **opMultiplication**.
Expresión regular / → **/** Token: **opDivision**.
Expresión regular && → **& &** Token: **opLogicAnd**.
Expresión regular || → **| |** Token: **opLogicOr**.
Expresión regular % → **%** Token: **opModule**.

Token asignación

El lenguaje soporta asignación simple como compuesta.

Expresión regular = → **=** Token: **assignment**.
Expresión regular += → **+ =** Token: **assignmentAddition**.
Expresión regular -= → **- =** Token: **assignmentSubtraction**.

Tipos de errores léxicos

LexicalExceptionGeneric

Se produce este error, cuando el primer caracter del próximo token es uno que no coincide con ninguno de los caracteres iniciales de los posibles tokens, en otras palabras, se encontró probablemente un caracter que no pertenece al lenguaje.

LexicalExceptionCommentary

Corresponde al error léxico donde un comentario multilínea no fue cerrado correctamente, o en otras palabras, llegó al EOF y no encontró la secuencia de caracteres de cierre. Este tipo de error es imposible que ocurra para comentarios de una línea, ya que un único '/' corresponde al operador y si ingresa un segundo '/' no importan los caracteres que posea a la derecha, siempre que llegue a un fin de línea o un EOF será válido.

LexicalExceptionLiteralCharacterEmpty

El error está asociado a la definición de un caracter vacío, de la forma '' (dos comillas simples consecutivas). Si o si, debe definirse un caracter dentro de las comillas simples.

LexicalExceptionLiteralCharacterNotClosed

Ocurre cuando dentro de un caracter se realiza un salto de línea o bien se encuentra un EOF. También ocurre cuando no se cierra debidamente el caracter con la segunda comilla simple.

LexicalExceptionString

Los string no deben ser interrumpidos por un salto de línea, por lo que si detecta uno antes de encontrar la comilla doble de cierre caerá en este error. La otra posibilidad de que este error ocurra, es que antes de encontrar la comilla doble de cierre, se encuentre el EOF del archivo.

LexicalExceptionUnicode (LOGRO)

Una vez que se empezó a leer un caracter y se forma el lexema '\u, estamos al tanto de que lo que sigue son 4 dígitos en hexadecimal con una comilla simple al final. Si hay menos de 4 dígitos en hexa, o bien, no hay una comilla simple al final, o bien en el lugar de algunos de los 4 digitos recibe un caracter no válido, ocurrirá este error léxico.

LexicalExceptionLogicAnd

Este error se detecta cuando luego de un caracter ‘ & ’ no hay otro igual seguido, es decir, si la cadena no está formada por dos ‘ & ‘ consecutivos de la forma ‘ && ‘, luego de leer el primero ocurre este error.

LexicalExceptionLogicOr

Este error es equivalente al del AND lógico, con la diferencia de que en lugar de detectar el error cuando no hay dos ‘ & ‘ seguidos sin espacio, lo detecta cuando no hay dos ‘ | ‘ seguidos sin espacio.

LexicalExceptionInteger

Nuestro lenguaje, MiniJava, solo soporta números de hasta nueve dígitos, por lo que detectar uno con diez o más, provocará un error léxico de este tipo.

Decisiones de diseño

1. **Implementación del autómata**

   Para llevar a código el autómata utilizó la estrategia MetodoXEstado, donde cada estado deL autómata es representado por un método en la clase LexicalAnalizer y la obtención del token se realiza mediante llamadas recursivas entre esos métodos.

2. **Palabras clave**

   Para identificarlas, decidí que una propuesta más inclinada por clean code, sería mantener una clase con un mapeo que poseen las palabras clave, y que mediante un booleano informe si la palabra solicitada pertenece al mapeo o no. La llamada al método de la clase ReservedWords es realizada por LexicalAnalizer, únicamente con aquellos tokens que pertenecen a un **idMetVar**, ya que las palabras clave no arrancan con mayúsculas.

3. **Caracter '\r'**

   Debido a que en algunos casos volvía el puntero al inicio de la línea, y me generaba errores al momento de mostrar en detalle los errores léxicos, decidí que el administrador de archivos saltee los caracteres \r y retorne el caracter siguiente.

4. **Literales enteros**

   Con respecto a estos tokens, decidí que si recibo una cadena de dígitos que supera el máximo permitido (nueve), voy a leer la cadena por completa como si fuese un único error, pero el lexema será formado de igual manera que con los demás errores, es decir, en el lexema se podrá leer hasta el último dígito válido. Mientras que en el detalle del error, se señalará el primer caracter de la línea que provocó el error.

5. **Estados de error**

   Considero que reportar en un estado en particular los errores sería más acorde a la estructura que de implementación del autómata que seguí, pero implementar en código ese estado sería un trabajo más complicado, ya que debería ver que tipo de errores debo lanzar, o bien crearlo en el estado anterior y brindarselo al estado de error, por lo que sería el único estado que reciba parametros y eso no me parecía del todo buena idea.

6. Error elegante

En mi caso, se indica la posición del error en el lugar donde se detectó, es decir, una posición más adelante de hasta donde se armó el lexema ya que, el lexema contiene toda la cadena que venía correspondiendose con un token de forma válida, y cuando lee un caracter que rompe este formato, el lexema deja de armarse pero se utiliza la última posición de columna para indicar el error ya que ahí debería haber otra cosa.

Las excepciones a esta regla son los errores correspondientes a comentarios multilineas y las correspondientes a errores de números enteros.

7. Error elegante excepciones

En los comentarios multilineas, el puntero que se utiliza para señalar la posición del error indica el caracter inicial '/' en el que inicia el comentario.

En los números enteros que poseen más de 9 dígitos, el puntero para señalar la posición del error indica el dígito en la posición 10, aquel que no corresponde a la cantidad máxima de dígitos permitida. Además el lexema para este caso, contiene únicamente 9 dígitos.

8. Strings

Tome la decisión de no seguir exactamente las indicaciones de los aspectos léxicos de MiniJava, y resolver lo siguiente como se resuelve en Java. En el informe de aspectos léxicos se nos plantea que una barra invertida no puede anteceder a la comilla doble de cierre del String, pero en Java esto es posible si es son dos barras invertidas seguidas, ya que la segunda pierde el significado que posee y pasa a ser un caracter más.

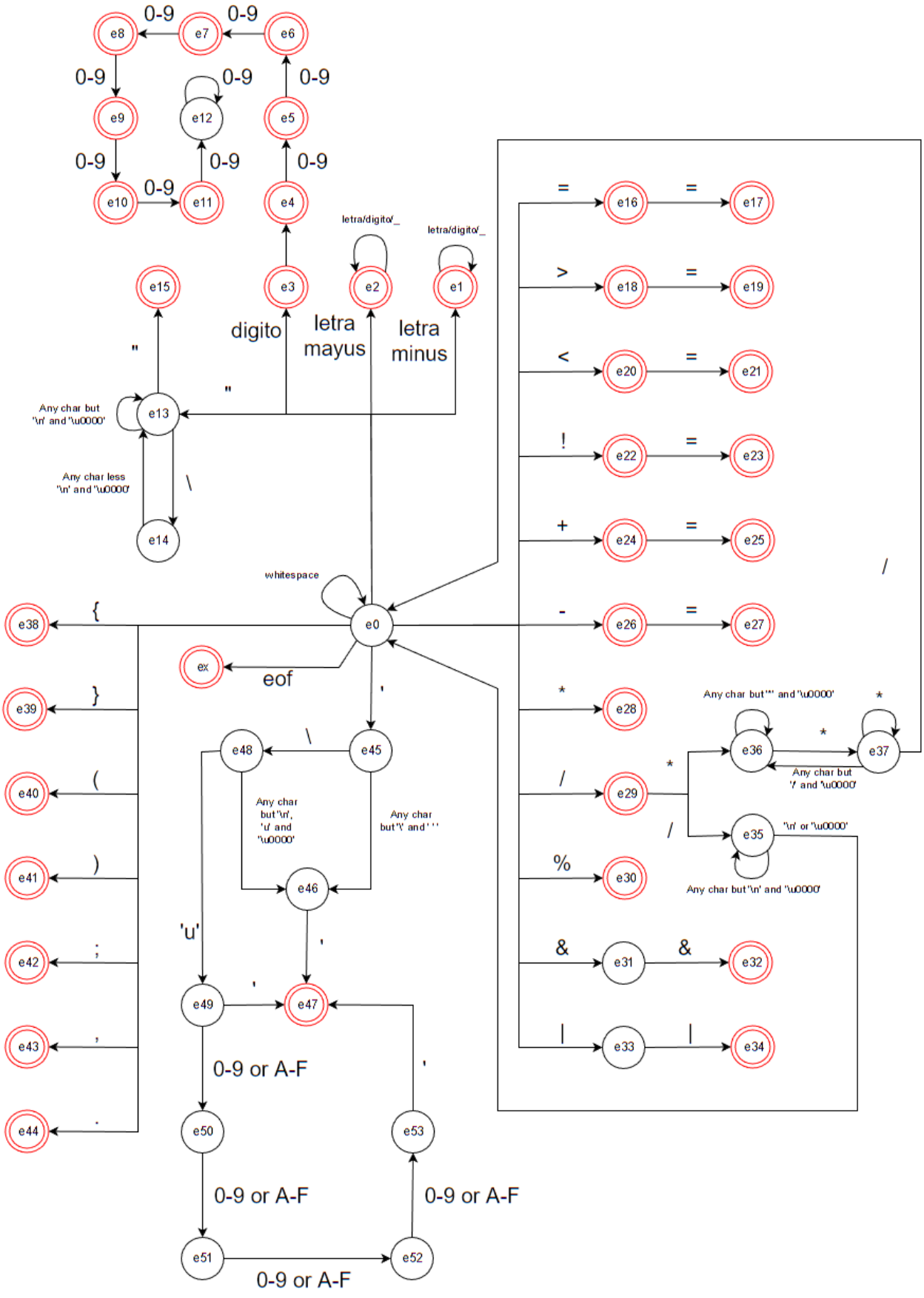
De esta manera permito Strings que contienen una cantidad par de barras invertidas de manera consecutiva antes de la comilla doble de cierre.

9. Detección de error en caracteres

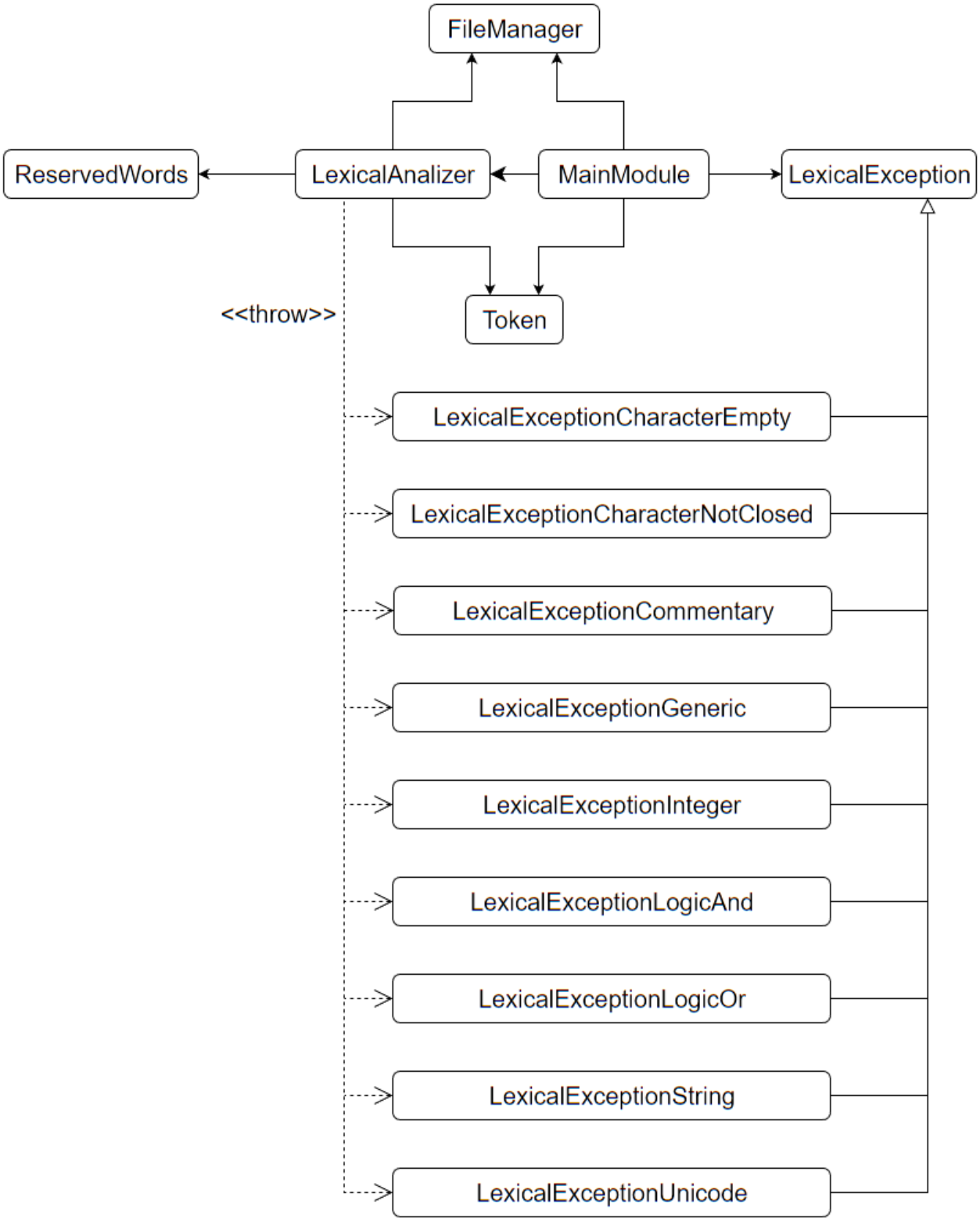
Cuando en Java olvidamos de cerrar un caracter, el error se extiende a lo largo de toda la línea de código, pero para mí implementación particular, el error es hasta que se detecta un caracter que no corresponde a un token válido del lenguaje, por ejemplo:

'\u0R_ejemplo → [Error:'\u0|línea] y por separado (idClass, R_ejemplo, línea)

Máquina de estados



Clases


FileManager
ReservedWords
LexicalAnalizer
MainModule
LexicalException
<<throw>>
Token
LexicalExceptionCharacterEmpty
LexicalExceptionCharacterNotClosed
LexicalExceptionCommentary
LexicalExceptionGeneric
LexicalExceptionInteger
LexicalExceptionLogicAnd
LexicalExceptionLogicOr
LexicalExceptionString
LexicalExceptionUnicode

Logros buscados

Entrega Anticipada Léxica

(no valido para la reentrega, ni etapas subsiguiente)

La entrega debe realizarse 48hs antes de la fecha limite

Imbatibilidad Léxica

(no valido para la reentrega, ni etapas subsiguiente)

El software entregado pasa correctamente toda la batería de prueba utilizada por la cátedra

Chars en Unicode

El Analizador Léxico permite literales carácter utilizando el el código unicode. Esto se hace mediante el escape \u seguido de un numero hexadecimal de 4 digitos. Por ejemplo '\u0A32' o '\uC90F'.

Reporte de Error Elegante

El compilador cuando reporta un error, además del numero de linea y la razón del error, muestra la linea en cuestión y apunta al lugar donde se produjo. Por ejemplo para el programa de la derecha debería mostrarse el mensaje:

```
Error Léxico en linea 2: # no es un símbolo valido
Detalle: v1 + # chau
             ^
[Error:#|2]
```

```
"hola"
v1 + # chau
if class}
```

Columnas

En los mensajes de error el compilador además del numero de linea indica el numero de columna donde se produjo el error. *Importante: esto afecta solo a los mensajes y no a los códigos de error!*

Multi-detección de Errores Lexicos

El compilador no finaliza la ejecución ante el primer error léxico (se recupera) y es capaz de reportar todos los errores léxico que tenga el programa fuente en una corrida

Chars en unicode

Como ya se vio anteriormente, se presentaron algunos aspectos de este logro junto con las condiciones básicas del analizador léxico, entre ellas, un tipo de error nuevo, su expresión regular y su parte del autómata que corresponde al estado 49, arrancando desde el autómata de los caracteres estándar en el 45.

Para implementarlo utilizó un estado intermedio, que detecta si el caracter es una simple '\u' y debe terminar la cadena de llamados en el estado finalizador de un caracter, o bien si es un caracter unicode y debe chequear si los siguientes cuatro caracteres son dígitos en hexadecimal, para luego terminar de igual manera en el estado final de un caracter estándar, si es que no ocurrio ningun tipo de error léxico en medio.

Reporte de error elegante

Para implementar esta funcionalidad, utilizo excepciones específicas, donde cada una posee un mensaje interno predefinido e información de la ubicación del error para poder más tarde brindarle la información almacenada al MainModule, para que formatee los mensajes de salida como el prefiera hacerlo, de manera que se pueda ver la línea completa de codigo, donde esta el error ubicado y el mensaje obligatorio de error con el lexema y su línea correspondiente.

Columnas

Para agregar esto al programa, únicamente se necesito agregar un atributo global a la clase que maneja el archivo (FileManager), de forma tal que maneje tanto las filas como las columnas del puntero en el archivo.

Multi-detección de errores léxicos

Para que se muestren todos los errores léxicos del archivo, lo que se hizo fue lo siguiente. Dentro del bucle while en el MainModule, que va solicitando los tokens hasta encontrarse con el token eof, se realiza el catch de los errores léxicos, de manera tal que luego de que ocurra un error, la ejecución del while prosiga.

Una **decisión de diseño** que tome frente a esta funcionalidad es que mi analizador léxico seguirá detectando tokens válidos antes, luego y entre los errores que detecte, y mientras notifica de tokens válidos irá reportando los errores léxicos que encuentre.