UNIVERSIDAD RAFAEL LANDIVAR, FACULTAD DE INGENIERÍA

# Proyecto (primera fase)

Ing. Msc. Víctor Orozco

13 de septiembre de 2022

## 1. INTRODUCCIÓN

Considerando la duración del curso en esta oportunidad se le presenta a usted el proyecto para la elaboración de un dialecto del lenguaje de programación COOL, denominado “CHILERO“.

Un dialecto de programación es una variación (relativamente pequeña) o una extensión de un Lenguaje de programación que no cambia su naturaleza intrínseca. En lenguajes como Scheme o Forth, las normas o estándares se puede considerar insuficientes, inadecuados o ilegítimos, incluso por los implementadores, por lo que a menudo se desvían de la norma o del estándar, creando un nuevo dialecto. En otros casos, un dialecto se crea para su uso en algunos Lenguaje específico del dominio, a menudo un subconjunto. Su dialecto sera una variante de COOL con palabras reservadas escritas en codificación es-GT.

En cada una de las fases, el estudiante cubrirá un componente del compilador: análisis léxico, análisis sintáctico y principios de análisis semántico. Cada asignación dará como resultado, en última instancia, una fase de trabajo del compilador que puede interactuar con otras fases.

Para esta primera fase, debe escribir un analizador léxico, también llamado escáner, utilizando para ello JFlex como generador de analizadores. Describirá el conjunto de tokens para “CHILERO“ en un formato de entrada apropiado, y el generador del analizador generará el código real para reconocer tokens en programas “CHILERO“.

# 2. INTRODUCCIÓN A JFLEX

JFlex permite implementar un analizador léxico escribiendo expresiones regulares y establecer acciones cuando alguna secuencia de caracteres coincide con alguna de estas. JFlex convierte estas reglas que ustedes van a definir en un archivo llamado lexer.lex en un archivo de Java con el código que implementa un autómata finito que puede reconocer estas expresiones regulares . Afortunadamente, no es necesario entender o incluso mirar lo que automáticamente genera JFlex. Los archivos que entiende JFlex están estructurados de la siguiente manera:

```
1 %{
2 Declaraciones
3 %}
4 Definiciones
5 %%
6 Reglas
7 %%
8 Subrutinas de Usuario
```

Las secciones de Declaraciones y Subrutinas de Usuario son opcionales y les permite a ustedes escribir declaraciones y funciones de ayuda en Java. Las sección de Definiciones también es opcional, pero en general es bastante útil porque las definiciones les permiten a ustedes darle nombres a las expresiones regulares. Por ejemplo, la siguiente definición:

```
1 DIGIT = [0-9]
```

le permite definir un dígito. Aquí, DIGIT es el nombre dado a la expresión regular que coincide con cualquier carácter entre 0 y 9. La siguiente tabla ofrece una descripción general de las expresiones regulares comunes que se puede especificar en Flex:

| | |
|---|---|
| x | el carácter "x" |
| "x" | una "x", incluso si x es un operador. |
| \x | una "x", incluso si x es un operador. |
| [xy] | el carácter x o y. |
| [x-z] | los caracteres x, y o z. |
| [^ x] | cualquier carácter excepto x. |
| . | cualquier carácter excepto nueva línea. |
| ^ x | una x al principio de una línea. |
| <y>x | una x cuando Lex está en la condición de inicio y. |
| x$ | una x al final de una línea. |
| x? | una x opcional |
| x* | 0,1,2, ... instancias de x. |
| x+ | 1,2,3, ... instancias de x. |
| x\|y | una x o una y. |
| (x) | una x |
| x/y | una x pero solo si es seguida por y. |
| {xx} | la traducción de xx de la sección de definiciones (macros). |
| x{m,n} | m hasta n ocurrencias de x |

La parte más importante de su analizador léxico es la sección de reglas. Una regla en JLex especifica una acción a tomar si la entrada coincide con la expresión regular o definición al principio de la regla. La acción a tomar es especificada escribiendo código de Java regular. Por ejemplo, asumiendo que un dígito representa un token en nuestro lenguaje (noten que este no es el caso de COOL), la regla sería entonces:

```
1 {DIGIT} {
2   AbstractSymbol num = AbstractTable.inttable.addString(yytext()
      );
3   return new Symbol(TokenConstants.INT_CONST, num);
4 }
```

Esta regla guarda el valor del dígito en una variable global AbstractTable.inttable y retorna el código apropiado para el token.

Un punto importante a recordar es que la entrada actual (es decir, el resultado de llamar a la función `next_token()` puede coincidir con múltiples reglas, JLex toma la regla que coincide con el mayor número de caracteres (maximal munch). Por ejemplo, si ustedes definieran las siguientes dos reglas:

```
1 [0-9]+      { // action 1 }
2 [0-9a-z]+   { // action 2 }
```

y si la secuencia de caracteres "2a.ªparece en el archivo que está siendo analizado, entonces la acción 2 va a tomarse, dado que la segunda regla coincide con más caracteres que la primer regla. Si múltiples reglas coinciden con la misma cantidad de caracteres, entonces la regla que aparece primero es la que se toma. En resumen los DFA generados con JFlex traba-

jan bajo los principios “maximal munch“ y “prioridad“.

Al escribir reglas en Flex, puede ser necesario realizar diferentes acciones dependiendo de los tokens previamente encontrados. Por ejemplo, al procesar un token de comentario de cierre, es posible que le interese en saber si se encontró previamente un comentario de apertura. Una forma obvia de rastrear el estado es declarar variables globales en su sección de declaración, las cuales (en forma de bandera) pueden pasar a `true` cuando ciertos tokens de interes son encontrados. JFlex también proporciona azúcar sintáctico para lograr una funcionalidad similar mediante el uso de declaraciones de estado tales como:

```
1 %state COMMENT
```

que puede ser cambiado a true escribiendo `yybegin(COMMENT)`. Para tomar una acción si y solo si un token que representa abrir comentario ha sido encontrado anteriormente. Pueden agregarle un predicado a su regla utilizando la siguiente sintáxis

```
1 <COMMENT> reg_exp {
2   // action
3 }
```

Hay un estado por defecto llamado `YINITIAL` que está activo a menos que ustedes explícitamente indiquen el inicio de un nuevo estado utilizando `yybegin(STATE)`. Ustedes pueden encontrar útil esta sintaxis para varios aspectos de esta asignación, así como reportar errores. Esta y otras particularidades de JFlex se describen con mejor detalle en la documentación oficial del proyecto.

## 3. Resolución

Para iniciar la resolución de esta primera fase su intructor le proporcionara un enlace hacia un repositorio en GitHub Classroom.

## 4. Lexer

En esta primera fase se espera que escriba reglas Flex que coincidan con las expresiones regulares apropiadas para definir tokens válidos en Cool como se describe en la Sección 10 y la Figura 1 del manual Cool y realizar el acciones apropiadas, como devolver un token del tipo correcto, registrar el valor de un lexema, y traducir el lexema original hacia un lexema compatible con CUP en inglés (donde sea apropiado), o informar un error cuando se encuentra un error.

Antes de comenzar con esta tarea, asegúrese de leer la Sección 10 y la Figura 1 del manual Cool; luego estudie los diferentes tokens definidos en `TokenConstants.java`. Su implementación necesita definir reglas JFlex que coincidan con las expresiones regulares definiendo cada token.

Por ejemplo, si coincide con un token `BOOL_CONST`, su lexer debe registrar si su valor es verdadero o falso; De manera similar, si coincide con un token `TYPEID`, debe registrar el nombre del tipo. Tenga en cuenta que no todos los tokens requieren almacenar información adicional; por ejemplo, solo devolver el tipo de token es suficiente para algunos tokens como palabras clave.

Su escáner debe ser robusto, debe funcionar para cualquier entrada concebible. Por ejemplo, debe manejar errores como un EOF que ocurre en medio de una cadena o comentario, así como constantes de cadena que son demasiado largos. Estos son solo algunos de los errores que pueden ocurrir; ver el manual para el resto. Debe hacer alguna provisión para la terminación correcta si ocurre un error fatal.

Para esta fase sera inaceptable si el programa finaliza su ejecución de forma abrupta o si no es capaz de capturar alguna excepción.

### 4.1. ERRORES

Todos los errores deberían de ser pasados al parser. Su lexer no debería de imprimir NADA. Los errores se comunican al parser retornando un token de error especial llamado `ERROR` junto con el mensaje de error.

Ustedes deben de ignorar el token error definido tambien en TokenConstants.java y no el token ERROR para esta asignación ya que este es utilizado únicamente en el parser.

Hay varios requerimientos para reportar errores y recuperarse de errores léxicos:

- Cuando un caracter inválido (alguno que no puede ser algún token) se encuentra, un string que contenga solo ese caracter debería de ser retornado como el error. Tienen que resumir el análisis con el siguiente caracter.
- Si un string contiene un newline sin escape, tienen que reportar el error como “Unterminated string constant“ y resumir el análisis léxico al principio de la siguiente línea. Esto quiere decir que estamos asumiendo que el usuario simplemente olvido cerrar el string con una comilla.
- Cuando un string es demasiado largo, tienen que reportar el error “String constant too long“. Si el string contiene caracteres inválidos (por ejemplo, el caracter nulo), tienen que reportar esto como “String contains null character“. En cualquier caso, el análisis debería de continuar hasta el final del string. El final del string es definido tanto como:
    - El principio de la siguiente línea si un newline es encontrado después de encontrar el error.
    - Después de cerrar el string con ".

- Si un comentario se queda abierto y se encuentra el caracter EOF, se tiene que reportar este error con “EOF in comment“. Por favor no tokenizen el contenido de los comentarios simplemente porque no se ha cerrado. Similarmente con los strings, si un EOF es encontrado, reporten el error como .EOF in string constant".
- Si se encuentra un *) fuera de un comentario, tienen que reportar el error como Ünmatched *)", en vez de tokenizar esta secuencia de caracteres como * y como ).

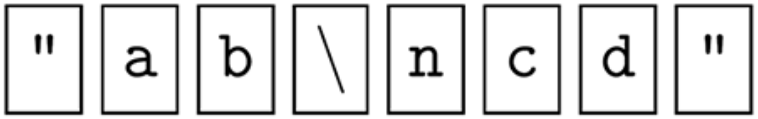
## 4.2. TABLA DE STRINGS

Los programas tienden a tener muchas ocurrencias del mismo lexema. Por ejemplo, un identificador es generalmente mencionado más de una vez en un programa (ąo de lo contrario no es muy útil!). Para ahorrar espacio y tiempo, una práctica común del compilador es almacenar lexemas en una tabla de cadenas. El proyecto base proporciona una implementación de tabla de cadenas para Java. Consulte las siguientes secciones para conocer los detalles.

Hay un problema al decidir cómo manejar los identificadores especiales para las clases básicas (Objeto, Int, Bool, String), `SELF_TYPE` y self. Sin embargo, este problema en realidad no surge hasta las fases posteriores de el compilador: el escáner debe tratar los identificadores especiales exactamente como cualquier otro identificador. No pruebe si los literales enteros se ajustan a la representación especificada en el manual Cool, simplemente cree un símbolo con el texto completo del literal como su contenido, independientemente de su longitud.

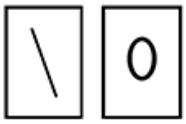
## 4.3. STRINGS

Su analizador léxico debería de convertir los caracteres que se les antepone un carácter de escape en las constantes string a sus valores correctos. Por ejemplo, si el programador escribe los siguientes ocho caracteres:

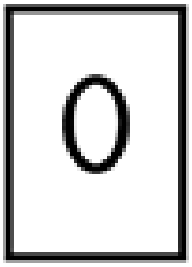



Su analizador léxico va a retornar un token `STR_CONST` cuyo valor semántico es estos 5 caracteres:

Donde \n representa el caracter de newline de la tabla ASCII. Siguiendo la especificación de la página 15 del manual de COOL, ustedes deberían de retornar un error para un string que contenga el caracter null. Sin embargo, la secuencia de estos dos caracteres:




Debería de ser convertida a un caracter:




### 4.4. OTRAS NOTAS

Su escáner debe mantener la variable `curr_lineno` que indica qué línea en el texto de origen es actualmente siendo escaneado. Esta función ayudará al analizador a imprimir mensajes de error útiles. Debe ignorar el token `LET_STMT`. Solo lo utiliza el parser. Finalmente, tenga en cuenta que si la especificación léxica está incompleta (alguna entrada no tiene una expresión regular que coincida), entonces el los escáneres generados por flex y jlex hacen cosas indeseables. Asegúrese de que su especificación esté completa

## 5. DIALECTO

En la siguiente tabla usted encontrara los lexemas equivalentes para COOL, denominado “CHILERO“, su analizador léxico debe reconocer los lexemas en español (CHILERO) y traducirlos al idioma inglés. Así mismo debe reconocer los lexemas originales en ingles, e incluso reconocer los lexemas en modo spanglish (es decir un programa que combina ambos lenguajes):

| Inglés | Español |
|---|---|
| class | clase |
| else | delocontrario |
| false | falso |
| fi | is |
| if | si |
| in | en |
| inherits | hereda |
| isvoid | esvacio |
| let | lavar |
| loop | ciclo |
| pool | olcic |
| then | entonces |
| while | mientras |
| case | encaso |
| esac | osacne |
| new | nuevo |
| of | de |
| not | nel |
| true | verdadero |

## 6. NOTAS ADICIONALES

- Cada llamada al lexer retorna el siguiente token y lexema de la entrada. El valor retornado por el método `CoolLexer.next_token` es un objeto de la clase `java_cup.runtime.Symbol`. Este objeto tiene un campo que representa el tipo del token (por ejemplo si es un entero, punto y coma, dos puntos, etc). Los códigos sintácticos o los tipos de cada token están definidos en el archivo `TokenConstants.java`. El componente, el valor semántico o lexema (si el token tiene), también es colocado en el objeto `java_cup.runtime.Symbol`. La documentación para la clase `java_cup.runtime.Symbol` está disponible en las referencias. Algunos ejemplos también están en el archivo base.

## 7. REFERENCIAS


1. Manual de JFlex
2. Manual de COOL
3. Manual de CUP


## 8. NOTAS FINALES

Fecha límite de entrega: Viernes 23 de septiembre, 23:55