Análisis Léxico.
Un Scanner para MiniLan (I)

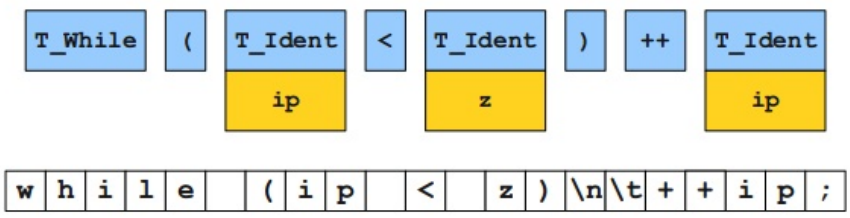
Autómatas y Matemáticas Discretas
Escuela de Ingeniería Informática
Universidad de Oviedo
Curso 2017-2018

1. Resumen de la sesión anterior

1.1. Análisis Léxico

Un *analizador léxico* o *scanner* lee un archivo de entrada y lo trocea en cadenas relevantes, estando cada una de estas cadenas relacionada con una categoría predefinida. La salida del analizador léxico será la secuencia de tokens que alimentara al *analizador sintáctico* o *parser*. Un token está formado por el par *nombre del token* y (posiblemente) el valor del mismo: p.e., <`T_Ident`, `sumatorio`>. El primer elemento es la categoría correspondiente de entre las predefinidas, el segundo provee información relacionada con la cadena encontrada (en el ejemplo anterior, la propia cadena).

La siguiente figura ilustra el proceso. Si se lee un código correspondiente al bucle while de la parte inferior de la figura, la secuencia de caracteres a leer es la mostrada en la parte intermedia, mientras que los tokens se muestran coloreados en la parte superior.




```
while (ip < z)
    ++ip;
```

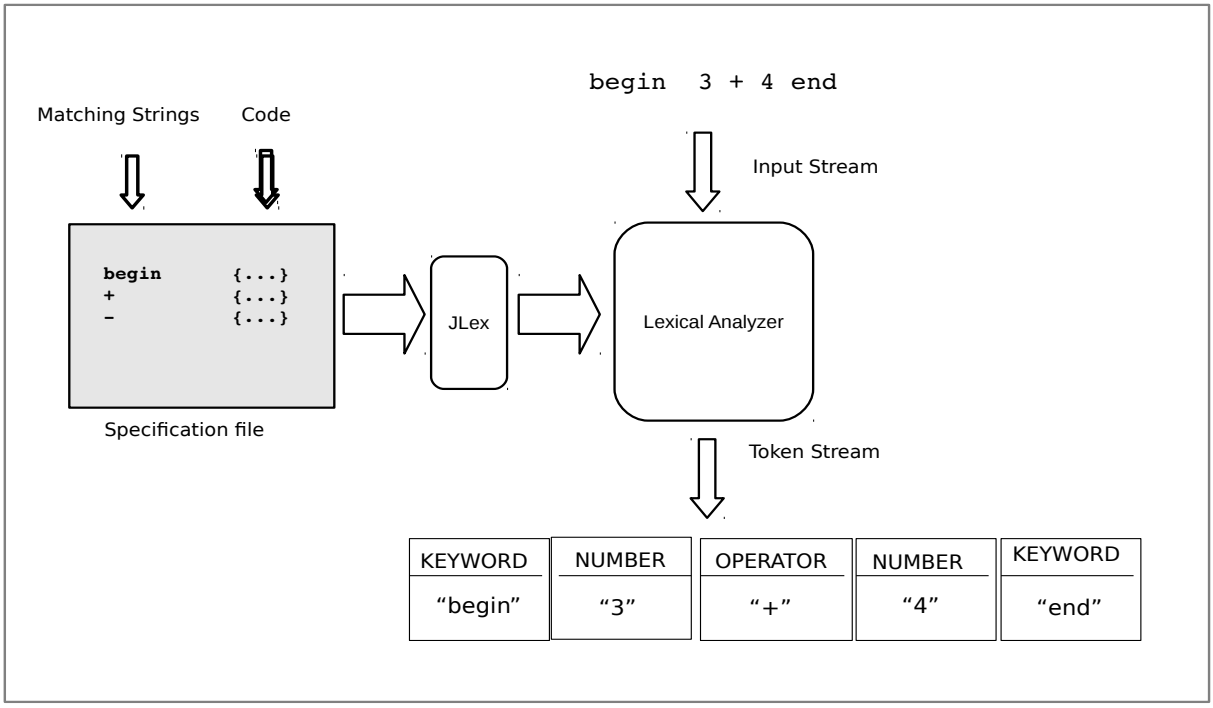
(Keith Schwarz)

1.2. Implementación de analizadores léxicos usando la familia de herramientas Lex

Dada la complejidad para implementar analizadores léxicos, existen herramientas que permiten y facilitan esta tarea, herramientas conocidas como **generadores de analizadores léxicos**. Básicamente, estas herramientas reciben una **especificación de los tokens**, normalmente en notación de **expresiones regulares**,

y generan el código fuente del analizador léxico correspondiente. Un ejemplo de estas herramientas es **JLex**, que es la que usaremos en esta asignatura; los analizadores que genera son programas en Java.

El proceso completo se puede esquematizar como se indica en la siguiente figura.



En otras palabras, hay que seguir los siguientes pasos:

1. Definir el conjunto de tokens (p.e., números enteros) y su descripción precisa (p.e., secuencia de uno o más dígitos).
2. Determinar la expresión regular para este token e incluirlo en el archivo de especificación.
3. Generar el scanner para las especificaciones incluidas y comprobar su validez.

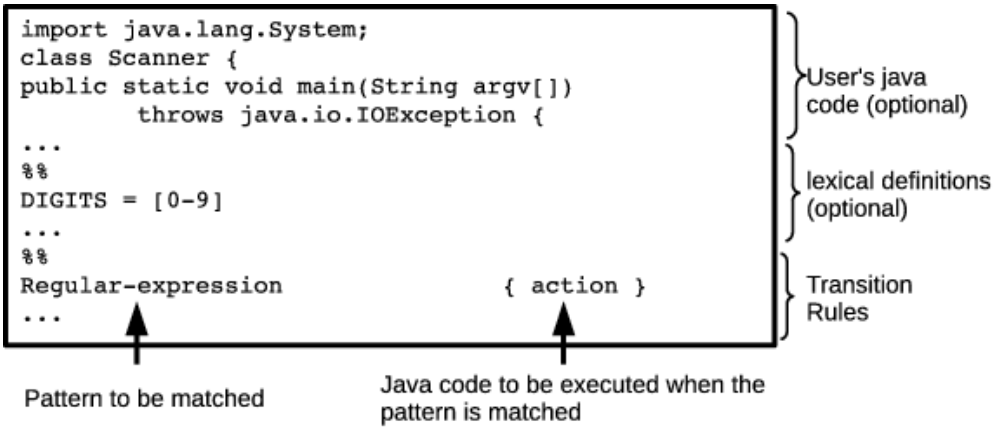
2. Determinar el fichero de especificación

Primeramente, hay que determinar el lenguaje para el cuál se genera el analizador léxico. El equipo docente ha desarrollado una descripción de las cadenas propias del un pequeño lenguaje que hemos denominado MiniLan.

La meta es escribir el archivo de especificación, en formato que entienda el JLex, que representa estas cadenas y que permita generar el scanner.

El formato del archivo de especificación JLex se puede observar en la siguiente figura. Está dividido en tres secciones (código de usuario, definiciones léxicas y reglas de transición) separadas por líneas conteniendo

un doble carácter porcentaje %%. Éstos deben estar situados al inicio de la línea, descartándose el resto de caracteres que pueda haber en dicha línea.



2.1. Sección de código de usuario

Se trata de un código fuente en lenguaje Java que debe estar libre de errores. Este código se copia directamente al código del scanner que vamos a generar. A lo largo de este curso, el código a utilizar en esta sección será proporcionado por el equipo docente cuando sea necesario.

2.2. Sección de Reglas de emparejamiento con expresiones regulares

Esta sección, la tercera, contiene las reglas que describen los patrones de las secuencias de caracteres o cadenas a reconocer, especificando la acción a realizar para cada una de ellas. Es decir, que esta sección tiene el siguiente formato, donde *patrónX* está definido mediante expresiones regulares y *accionX* es código en Java entre llaves ({}):

```
patrón1 accion1
patrón2 accion2
...
```

Cuando una secuencia de caracteres leída se corresponde con un patrón de los especificados se ejecuta la acción asociada a dicho patrón. Para acceder a la secuencia de caracteres que disparó la regla actual se debe utilizar la función `yytext()`.

2.2.1. Escribir expresiones regulares en JLex

El alfabeto para JLex es el conjunto de caracteres ASCII, es decir, caracteres codificados entre 0 y 127, ambos inclusive. De entre éstos, algunos de ellos tienen especial significación o función.

Las expresiones regulares en JLex no deben incluir ni espacios en blancos, ni tabulaciones. Estos caracteres se utilizan como finalizadores de las expresiones regulares.

Los siguientes caracteres son meta-caracteres, es decir, tienen un significado especial -son operadores- dentro de las expresiones regulares de JLex:

```
$ ? * + | <( ) ^. [ ] { } " \ .
```

El resto de caracteres solo representan el propio carácter en sí mismo ('a', 'e', ...).

A continuación se describe cada uno de los operadores anteriores.

- `+`, `*` y `|` tienen el mismo significado dado en la clase de teoría (uno o más, cero o más, operador o). Ejemplo:

  `(0|1|2|3|4|5|6|7|8|9)` representa un dígito
  `(0|1|2|3|4|5|6|7|8|9)+` representa una secuencia de uno o más dígitos
  `(a|e|i|o|u)*` representa una secuencia cero o más vocales.

- `.` el punto representa cualquier carácter excepto el carácter nueva línea ($\backslash n$).

- `?` cero o una vez la expresión regular de la izquierda: `(0|1)?` denota un '0' o un '1' o nada(λ). En general: `regexp? == regexp | ` λ

- `[...]` los corchetes representan un conjunto de caracteres, es decir, que la expresión regular puede contener uno cualquiera de los caracteres contenidos en ese conjunto. Los conjuntos de caracteres se delimitan por corchetes; mientras que los caracteres individuales se indican sin comillas y/o separadores.

  ```
  [abc]==(a|b|c)
  [Aa]==(A|a)==(a|A)
  ```

  Para especificar un **rango** de caracteres se utiliza el carácter guión `-`

  ```
  [0-9]=(0|1|...|9)
  [a-z0-9]=(a|b|...|z|0|1....|9)
  ```

  Para especificar el complementario de un conjunto se usa el circunflejo (`^`)

  ```
  [^0-9] : denota cualquier carácter que no sea un dígito
  [^a-zA-Z] : denota cualquier carácter que no sea una letra
  ```

- Algunos caracteres (como los corchetes, llaves, <, `+`, `*`, o `.`) pierden su significado especial si le antecede el carácter `\` o se ponen entre dobles comillas `""`; en este caso, representan el correspondiente carácter. Por ejemplo: `"x*"` representa la secuencia de caracteres formada por una 'x' seguida por un '$*$'.

- `{name}` se reemplaza por el patrón definido en la sección de definiciones léxicas que veremos más adelante.

- Sensibilidad al contexto: JLex soporta reglas contextuales. Esto significa que la expresión regular no solo determina el tipo de cadena sino además dónde está situada dicha cadena:

  - `$`: si `$` es el último carácter en una expresión entonces esta expresión solo puede ser emparejada al final de una línea.
    `a$` emparejará una `"a"` si está al final de la línea, pero no en el caso de `aaab`.

  - `^`: si `^` es el primer carácter de una expresión entonces esta expresión solo puede ser emparejada al inicio de una línea.
    `^b` denotará una b al principio de la línea pero no si le precede algún carácter.

2.3. Sección de definiciones léxicas o *directivas*

Esta es la segunda sección del archivo de especificación JLex, que es donde se escriben las **definiciones de macro**. El propósito de las macros es el definir expresiones regulares que luego puedan ser reutilizadas.

El formato de una macro es como sigue. Cada definición de macro debe estar contenida en una línea independiente. Está formada por un nombre de macro al que le sigue un signo de asignación (=) y, finalmente, la **definición**.

```
nombreDeMacro = (definicionDeMacro)
```

Los nombres de las macros debe ser un identificador válido para Java, mientras que las definiciones de las macros deben ser expresiones regulares válidas. Muy importante: la definición de la macro **debe estar siempre escrita entre paréntesis**. No es que sea obligatorio, pero esta regla evitará errores posteriores. Sin embargo, es posible introducir espacios en blanco y tabuladores antes o después del signo igual de la definición.

Ejemplos, donde `DIGIT` es la macro que representa cadenas de caracteres compuestas por un único dígito, mientras que la macro `LETTER` representa cadenas de caracteres compuestas por una letra alfabética (bien en mayúsculas como en minúsculas).

```
DIGIT=([0-9])
LETTER=([a-zA-Z])
```

Una vez una macro se ha definido, es posible utilizarla en cualquier otra expresión regular, bien sea en el patrón de una regla de emparejamiento de la sección tercera del archivo de especificación, bien en otra macro. Para utilizar una macro solo es necesario introducir su nombre entre llaves: esto se denomina **expansión de la macro**.

Por ejemplo, dadas las definiciones de macro anteriores, los siguientes patrones pueden emparejarse con cualquier secuencia alfanumérica (letras o números) de al menos un carácter:

```
({LETTER}|{DIGIT})({LETTER}|{DIGIT})*
({LETTER}|{DIGIT})+
```

3. MiniLan: un MINI LANguage

MiniLan es el nombre de un lenguaje de programación de juguete que hemos creada para practicar este curso. La idea es definir durante este semestre el scanner y el parser para este lenguaje. La **descripción del vocabulario** del lenguaje MiniLan está disponible en Campus Virtual, será necesario consultarla durante todo el curso.

4. Ejercicios

4.1. Obtener los archivos

1. Muévete a tu directorio `~/practicasAMD/LexicalAnalysis/ScannerMiniLan` y copia los archivos fuente desde el directorio de la asignatura:
   `$cp /var/asignaturas/AMD/practicas/LexicalAnalysis/ScannerMiniLan/* .`
2. Comprueba que se dispone de los archivos:
   - `MiniLan.lex` con las especificaciones de **algunas** cadenas relevantes de **MiniLan**,
   - `generaScanner` para generar el scanner, como se hizo en la práctica anterior,
   - `test` para probar el scanner.

4.2. Expresiones regulares básicas

1. Muestra (*less*) el contenido del archivo **MiniLan.lex** y describe los cadenas a emparejar.
2. Genera el scanner: `$ sh generaScanner MiniLan.lex`

   La salida obtenida debe ser la siguiente, sino tienes algún error.

```
Processing first section -- user code.
Processing second section -- JLex declarations.
Processing third section -- lexical rules.
Creating NFA machine representation.
NFA comprised of 36 states.
Working on character classes.::::::::::::
NFA has 14 distinct character classes.
Creating DFA transition table.
Working on DFA states.................
Minimizing DFA transition table.
12 states after removal of redundant states.
Outputting lexical analyzer code.
```

3. Muestra (*less*) el contenido del archivo `test`
4. Comprueba el scanner con el archivo `test` : `$ java Scanner < test`

```
begin                                    SCANNER:: found Reserved Word BEGIN
;                                        SCANNER:: found punctuation symbol SEMICOLON
```

5. Edita el archivo `MiniLan.lex` y añade una línea después de la regla para el `begin`. Escribe la expresión regular para la palabra clave `end`, cuya acción es imprimir el mensaje `SCANNER:: found Reserved Word END`.
6. Genera el scanner de nuevo. Edita el archivo de test para añadir una cadena `end` donde desees.
7. Ejecuta el scanner, ¿Qué ocurre?

4.3. Escribir macros

1. Escribe la macro `INTEGER` para denotar los enteros como una secuencia de uno o más dígitos. Utiliza para ello la macro `DIGIT` ya definida.
2. Escribe una nueva regla justo después de la regla para `DIGIT`. La expresión regular debe ser crearse utilizando la macro `INTEGER` y la acción asociada debe ser imprimir el mensaje `SCANNER::found NUMBER` $<$*el valor*$>$
3. Genera el scanner de nuevo. Edita el archivo `test` y añade la cadena `123  4` en la primera línea.
4. Ejecuta el scanner. ¿Qué ocurre ahora? ¿Es lo esperado?
5. Edita el archivo `MiniLan.lex` y cambia el orden de las reglas para las cadenas `INTEGER` y `DIGIT`. Genera el scanner y pruébalo con el archivo `test`. ¿Qué ocurre ahora?

5. Selección de disparo de reglas (I)

Si más de una regla puede ser disparada dada una determinada secuencia de caracteres de entrada, el scanner resuelve el conflicto seleccionando la regla que empareja la **cadena de mayor longitud**.

Si más de una regla presentan cadenas de igual longitud, el scanner selecciona la **regla que está definida primero en el archivo de especificación JLex**. Por lo tanto, las reglas que aparecen primero en el archivo de especificación tienen mayor prioridad.

Por ejemplo, supongamos se definen las reglas para `DIGIT` e `INTEGER` (en este order). Si la entrada es *33* el scanner selecciona la segunda regla; mientras que si la entrada es *3* entonces la regla seleccionada será la primera.

6. Ejercicios

1. Edita el archivo `test` y añade el carácter `-` en la primera línea.
2. Genera y ejecuta el scanner con el archive de test. ¿Qué ocurre a hora?¿Por qué?

7. Selección de disparo de reglas (II)

Las reglas definidas en el archivo de especificación JLex **deben emparejar toda posible entrada**: si esto no ocurre, entonces el scanner generará un error cuando se encuentre una secuencia de caracteres no definida por el conjunto de reglas. Por lo tanto, toda posible entrada debe emparejarse con el patrón de al menos una regla. Para facilitar esta tarea se dispone de la última regla *comodín*, que empareja cualquier carácter no definido por las reglas anteriores:

```
.       {System.out.println("SCANNER:: Unmatched input <" + yytext() +">");}
```

El carácter punto (.), tal como se describe anterior regla, se empareja con cualquier entrada excepto el carácter nueva línea ($\backslash n$).

8. Ejercicios

1. Edita `MiniLan.lex` y añade una nueva línea **al final del archivo**. Escribe la expresión regular para el operador `-` que imprima el mensaje `SCANNER:: found Operator MINUS`.
2. Genera y ejecuta le scanner de nuevo. ¿Qué ocurre? ¿Por qué?
3. Resuelve el problema, genera el scanner y comprueba que se detecta el operador indicado.

9. Ejercicios finales

1. Modifica el archivo de especificación para añadir dos nuevos operadores:
   a) Multiplicación: el símbolo del operador es el carácter `*` y la salida debe ser el mensaje: `SCANNER:: found Operator MULT`.
   b) División. El símbolo del operador es el carácter `/` y la salida debe ser el mensaje: `SCANNER:: found Operator DIV`.
2. Identificar los paréntesis de apertura y cierre: para la apertura de paréntesis se mostrará el mensaje `SCANNER:: found Symbol LP`, mientras que para el cierre se debe mostrar `SCANNER:: found Symbol RP`.
3. Añadir la regla para la palabra reservada de lenguaje `print`, con la acción: `SCANNER:: found Reserved Word PRINT`.
4. Modificar el archivo de test para comprobar que las definiciones realizadas son correctas. Se debe comprobar TODAS las reglas introducidas.
5. Elimina la regla asociada a la macro `DIGIT` y deja únicamente la asociada a `INTEGER`