Lexer (Expresiones Regulares y Autómatas Finitos)

Volvamos al problema original

Para parsear el texto de entrada:

```
let x=read() in 2*(x+5)
```

El primer paso es convertir a una secuencia de tokens:

```
let x = read ( ) in 2 * ( x + 5 )
```

¿Qué son los tokens?

```python
class Token:
    def __init__(self, line, column, type, lexeme):
        self.line = line
        self.column = column
        self.type = type
        self.lexeme = lexeme

>>> tokenize("let x=read() in 2*(x+5)")
[
    Token(line=1,column=1,type="let",lexeme="let"),
    ## ...
    Token(line=1,column=5,type="id",lexeme="x"),
    ## ...
    Token(line=1,column=16,type="number",lexeme="2"),
    ## ...
]
```

¿Por qué tokenizar?

Para simplicar la gramática:

- Normalizar los literales, identificadores, etc.
- Eliminar los espacios en blanco innecesarios.
- Eliminar los comentarios.
- Reemplazar caracteres especiales.
- Lidiar con los detalles del *encoding*.
- Detectar errores léxicos (e.j. `string` incompletos).

Haciendo un tokenizador

¿Podemos hacer una gramática LL para los tokens?

```
digit   -> '0' | '1' |...| '9'
number  -> digit | digit number
char    -> 'a' | ... | 'z' | 'A' | ...
alpha   -> char alpha | num alpha | epsilon
id      -> char alpha
...
lparen  -> '('
rparen  -> ')'
oper    -> '*' | '/' | '-' | '+'
...
let     -> 'l' 'e' 't'
...
ws      -> ' ' | '\t' | '\n'
comment -> ...
```

¿Qué problemas tiene este enfoque?

- ¿Cuántos tokens tienen prefijos comunes?
- ¿Qué tamaño tendrá el árbol de derivación?
- ¿Cómo son las reglas semánticas de cada token?
- ¿Cómo logro el comportamiento *greedy*?
- ¿Podemos resolverlo con *menos* que LL?

```
    Input:  |l e t  x = r e a d ( )  i n  2 * ( x + 5 )
    Token:
    Output:
```

```
    Input:   l|e t  x = r e a d ( )  i n  2 * ( x + 5 )
    Token:   l
    Output:
```

```
Input:   l e|t  x = r e a d ( )  i n  2 * ( x + 5 )
Token:   le
Output:
```

```
Input:   l e t| x = r e a d ( )  i n  2 * ( x + 5 )
Token:   let
Output:
```

```
Input:   l e t |x = r e a d ( )  i n  2 * ( x + 5 )
Token:
Output:  let
```

```
Input:   l e t  x|= r e a d ( )  i n  2 * ( x + 5 )
Token:   x
Output:  let
```

```
Input:   l e t  x =|r e a d ( )  i n  2 * ( x + 5 )
Token:   =
Output:  let x
```

```
Input:   l e t  x = r|e a d ( )  i n  2 * ( x + 5 )
Token:   r
Output:  let x =
```

```
Input:   l e t  x = r e|a d ( )  i n  2 * ( x + 5 )
Token:   re
Output:  let x =
```

```
Input:   l e t  x = r e a|d ( )  i n  2 * ( x + 5 )
Token:   rea
Output:  let x =
```

```
Input:   l e t  x = r e a d|( )  i n  2 * ( x + 5 )
Token:   read
Output:  let x =
```

```
Input:   l e t  x = r e a d (|)  i n  2 * ( x + 5 )
Token:   (
Output:  let x = read
```

```
Input:   l e t  x = r e a d ( )| i n  2 * ( x + 5 )
Token:   )
Output:  let x = read (
```

```
Input:   l e t  x = r e a d ( ) |i n  2 * ( x + 5 )
Token:
Output:  let x = read ( )
```

```
Input:   l e t  x = r e a d ( )  i|n  2 * ( x + 5 )
Token:   i
Output:  let x = read ( )
```

```
Input:   l e t  x = r e a d ( )  i n| 2 * ( x + 5 )
Token:   in
Output:  let x = read ( )
```

```
Input:   l e t  x = r e a d ( )  i n |2 * ( x + 5 )
Token:
Output:  let x = read ( ) in
```

```
Input:   l e t  x = r e a d ( )  i n  2|* ( x + 5 )
Token:   2
Output:  let x = read ( ) in
```

```
Input:   l e t  x = r e a d ( )  i n  2 *|( x + 5 )
Token:   *
Output:  let x = read ( ) in 2
```

```
Input:   l e t  x = r e a d ( )  i n  2 * (|x + 5 )
Token:   (
Output:  let x = read ( ) in 2 *
```

```
Input:   l e t  x = r e a d ( )  i n  2 * ( x|+ 5 )
Token:   x
Output:  let x = read ( ) in 2 * (
```

```
Input:   l e t  x = r e a d ( )  i n  2 * ( x +|5 )
Token:   +
Output:  let x = read ( ) in 2 * ( x
```

```
Input:   l e t  x = r e a d ( )  i n  2 * ( x + 5|)
Token:   5
Output:  let x = read ( ) in 2 * ( x +
```

```
Input:   l e t  x = r e a d ( )  i n  2 * ( x + 5 )|
Token:   )
Output:  let x = read ( ) in 2 * ( x + 5
```

```
Input:   l e t  x = r e a d ( )  i n  2 * ( x + 5 )|
Token:
Output:  let x = read ( ) in 2 * ( x + 5 )
```

Haciendo un tokenizador v2.0

```python
def tokenize(input):
    state = 0
    curr = ""
    for c in input:
        if state == 0:    ## None
            if c.isalpha():
                curr += c
                state = 1 ## id | keyword
            elif c.isdigit():
                curr += c
                state = 2 ## constante
            ## ...
        if state == 1:    ## id | keyword
            if c.isspace() or c in oper:
                if curr == 'let':
                    yield Token(type='let', ...)
                    ## ...
```

Automatizando este proceso

Tendremos 2 componentes fundamentales:

- Una componente *declarativa*, de "alto nivel" para definir cómo lucen sintácticamente los tokens.

- Una componente *procedural*, de "bajo nivel" que reconocerá automáticamente los tokens.

Y como ya es usual, diseñaremos un mecanismo para convertir de "alto nivel" a "bajo nivel".

Expresiones regulares

Una expresión regular es una definición recursiva de un lenguaje:

- a es la expresión regular para $L(a) = \{a\}$
- ϵ es la expresión regular para $L(\epsilon) = \{\epsilon\}$

Si s y r son expresiones regulares, entonces:

- **Unión**: $(s)|(r)$ es la expresión regular para el lenguaje $L(s) \bigcup L(r)$.
- **Concatenación**: $(s)(r)$ es la expresión regular para el lenguaje $L(s)L(r)$.
- **Clausura**: $(s)*$ es la expresión regular para el lenguaje $L(s)* = \bigcup_{k=0}^{\infty} L(s)^k$.

Un lenguaje definido de esta forma, le llamaremos **lenguaje regular**. Más adelante caracterizaremos formalmente a estos lenguajes.

Ejemplos

> - $(a|b)*$
> - $aa(a|b)*$
> - $(a|b)*bb$
> - $a*(baa*)*(b|\epsilon)$
> - $(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*$

Expresiones regulares extendidas

Sin cambiar la definición, podemos adicionar algunos operadores cómodos:

- **Opciones:** $[abc]$ será equivalente a $(a|b|c)$.
- **Rango:** $[a-z]$ (o cualquier otro rango sensible) será equivalente por $(a|b|...|z)$.
- **Cero o una vez:** $(s)?$ será equivalente a $(s)|\epsilon$.
- **Clausura positiva:** $(s)+$ será equivalente a $(s)(s)*$.

Además diremos que $*$ y $+$ tienen la mayor prioridad, y $|$ la menor y asocia a la izquierda.

NOTA: Al adicionar estos operadores no hemos cambiado la definición (ni el poder expresivo), luego toda demostración formal la podemos realizar solo con la definición original si es más cómodo.

El lenguaje de los tokens

Ahora podemos definir el lenguaje de los tokens más fácilmente:

$$\begin{array}{rr} id & [a-zA-Z][a-zA-Z0-9]* \\ num & [1-9][0-9]*(.[0-9]+)? \\ oper & [*/+-] \\ lparen & \backslash(\\ rparen & \backslash) \\ equals & = \\ let & \text{let} \\ in & \text{in} \end{array}$$

Demostrando equivalencias entre lenguajes

Si tenemos una expresión regular r y un lenguaje L, ¿cómo demostrar que $L(r) = L$?

- $L(r) \subseteq L$: simulando la expresión regular, demostrar que sólo reconoce cadenas correctas.
- $L \subseteq L(r)$: dar una forma de construir cada cadena de L.
- Apoyarse en cada sub-expresión s de r genera un sub-lenguaje $L(s)$, y construir recursivamente L aplicando las propiedades.

Ejemplo:

- $a*(baa*)*(b|\epsilon)$

Gramáticas regulares

Una **gramática regular** es una gramática $G =< T, N, P, S >$ donde todas las producciones tienen la forma:

- $A \ \rightarrow \ aB$
- $A \ \rightarrow \ a$
- $S \ \rightarrow \ \epsilon$

Siendo $a \in T$ un terminal y $A, B \in N$ no-terminales, y S el símbolo distinguido.

Ejemplo

$$\begin{array}{rcr} S & \rightarrow & aS \mid bA \mid \epsilon \\ A & \rightarrow & aS \end{array}$$

Algunos comentarios

Las expresiones regulares son la navaja suiza del procesamiento de texto:

- Buscar y reemplazar en archivos.
- Validar entradas en formularios.
- Extraer patrones en lenguaje casi natural.
- Syntax-highlight en editores de texto.
- ...

Que nos queda?

Necesitamos:

- Un **mecanismo reconocedor** (de bajo nivel)
- Que se obtenga automáticamente de una (*meta*) expresión regular
- Que permita distinguir cada token de entre los componentes
- Que sea resistente a errores

> Construir un reconocedor de lenguajes regulares.

Autómatas Finitos Deterministas (DFA)

Formalizando

Un DFA es un quíntuplo $A =< Q, q_0, V, F, f >$ donde:

- Q es un conjunto *finito* de estados ($Q = \{q_0, \dots, q_n\}$).
- $q_0 \in Q$ es el estado inicial.
- V es un conjunto finito de símbolos que pueden aparecer en la cinta.
- $F \subseteq Q$ es un subconjunto de estados que denominaremos *estados finales*.
- $f : Q \times V \rightarrow Q$ es una *función de transición*.

Lenguaje de un DFA

Definiremos $\mathrm{L}(A)$

Sea A un autómata, w una cadena, $\mathrm{Q} =< q^0, q^1, \ldots, q^{|w|} >$ una secuencia de $|w| + 1$ estados de Q, donde:

- $q^0 = q_0$
- $q^{i+1} = f(q^i, w_i)$

Diremos que A reconoce w, $w \in \mathrm{L}(A)$, si y solo si:

$$q^{|w|} \in F$$

> Notar que Q *es único* para una cadena w cualquiera. Es por esto que el autómata es **determinista**.

Autómata Finito No-Determinista

Formalizando

Un NFA es un quíntuplo $A =< Q, q_0, V, F, f >$ donde:

- Q es un conjunto *finito* de estados ($Q = \{q_0, \ldots, q_n\}$).
- $q_0 \in Q$ es el estado inicial.
- V es un conjunto finito de símbolos que pueden aparecer en la cinta.
- $F \subseteq Q$ es un subconjunto de estados que denominaremos *estados finales*.
- $f : Q \times V \bigcup \{\epsilon\} \to 2^Q$ es una *función de transición*.

NOTA: La diferencia es la función de transición, que permite saltar a más de un estado, y hacer ϵ -transiciones (sin leer de la cinta).

Lenguaje de un NFA

Definiremos $\mathrm{L}(A)$

Sea A un autómata, w una cadena, $\mathrm{Q} =< q^0, q^1, \ldots, q^k >$ una secuencia de $k \geq |w| + 1$ estados de Q, donde:

- $q^0 = q_0$
- $q^{i+j+1} = f(q^i, w_i)$, o
- $q^{i+j+1} = f(q^i, \epsilon)$ con $j \geq 0$.

Diremos que A reconoce w, $w \in \mathrm{L}(A)$, si y solo si, existe Q tal que:

$$q^k \in F$$

> Notar que Q *no es único* para una cadena w cualquiera. Es por esto que el autómata es **no-determinista**.

¿Es bueno el no-determinismo?

- Permite reconocer lenguajes más fácilmente (¿por qué?).
- Es más complicado de evaluar computacionalmente (¿por qué?).

Surgen entonces algunas preguntas

- ¿Son **más poderosos** los autómatas no-deterministas?
- ¿Existen lenguajes que un NFA puede reconocer, pero ningún DFA puede?
- O son los lenguajes reconocibles por los NFA también regulares?

Convirtiendo un NFA a un DFA

Definamos primero

Sea $Q' \subseteq Q$ un subconjunto de estados:

- $Goto(Q', c) = \{q_j \in Q \mid q_i \in Q',\ q_j \in f(q_i, c)\}$
- $\epsilon Closure(Q') = \{q_j \in Q \mid q_i \in \epsilon Closure(Q'),\ q_j \in f(q_i, \epsilon)\}$

Idea del algoritmo

- Los estados del nuevo DFA serán $Q' \subseteq Q$, cada estado es un elemento $Q' \in 2^Q$.
- Entre un par de estados Q_i, Q_j hay una transición con c, si y solo si, $Q_j = \epsilon Closure(Goto(Q_j, c))$.
- El estado inicial es $Q_0 = \epsilon Closure(q_0)$.
- Los estados finales son aquellos Q' que contienen un $q_i \in F$.

Consideraciones finales (por ahora)

El no-determinismo en lenguajes regulares no es un problema

- Se puede convertir a un DFA (aunque con una cantidad exponencial de estados)
- Se puede evaluar en tiempo lineal en la cadena (sin realizar la conversión completa)
- Brinda un poder expresivo que será útil para reconocer expresiones regulares