DLP

Introducción

Un procesador de lenguajes es una aplicación en la cual una de las entradas es un lenguaje.

Tipos de procesadores

- Traductor
- Compilador
- Intérprete

Fases de un traductor

ANALISIS LEXICO

Éste va leyendo de la entrada carácter a carácter hasta que detecta una cadena válida (lexema) o una cadena errónea.

Una vez encontrado un lexema, hace algo más: le asigna un número llamado Token (al cual se le puede ver como el tipo o la categoría a la que pertenece el lexema).

Un ejemplo de esto sería el siguiente:

LEXEMA	TOKEN
"print",	25
"while",	26
"..."	27

Salen parejas de (lexema, token).

Para representar dichos tokens habrá que realizar una transformación:

```
("print" , PRINT)
("while" , WHILE)
("return", RETURN)
```

```
class Léxico {
    static final int PRINT = 25;
    static final int WHILE = 26;
    static final int RETURN = 27;
    …
}
```

Para cada lexema de entrada se creara un objeto de la clase Token
el cual contendra:
- El lexema reconocido
- El tipo (categoria) del lexema (tokenType)
- La posicion del fichero (linea y columna)

En lugar de devolver la lista completa de tokens la forma de implementar esto será implementando un método en el léxico que pida el siguiente token disponible.
Para poder hacer que el programa tenga fin habrá que definir constantes:
- static final int END = 0;
- static final int WHILE = 257;
- static final int IF = 260;

EJEMPLO 0220

Supóngase un programa cuyo bucle principal de ejecución va invocando a nextToken y, a continuación, imprime una traza del contenido del objeto Token obtenido en cada llamada:

```
Lexico lex = new Lexico(new FileReader("prog.txt"));
Token token;
while ((token = lex.nextToken()).getType() ≠ Lexico.END) {
    System.out.println(token.getLexeme());
    System.out.println(token.getType());
    System.out.println(token.getLine());
}
```

Supóngase la siguiente entrada en el fichero prog.txt:

```
altura
+
1.75
```

Y finalmente supóngase que se han definido las siguientes constantes asociadas a los tokens del lenguaje:

```
class Lexico {
    static final int END = 0;
    static final int IDENT = 257;
```

```
        static final int IF = 258;
        static final int LITENT = 259;
        static final int LITREAL = 260;
        …
    }
```

El resultado de ejecutar dicho programa seria el siguiente:

Iteración	getLexeme()	getType()	getLine()
1	"altura"	257	1
2	"+"	43 ('+')	2
3	"1.75"	260	3

ESPECIFICACIÓN LÉXICA

1. Determinar tokens (como en el caso de los IDENT)
2. Definir patrones (determinar que lexemas pertenecen a cada token)

En el segundo caso determinar si un lexema es valido o no depende de una serie de reglas que el creador del lenguaje haya definido. Para definir estos patrones hay que hacer uso de los metalenguajes.

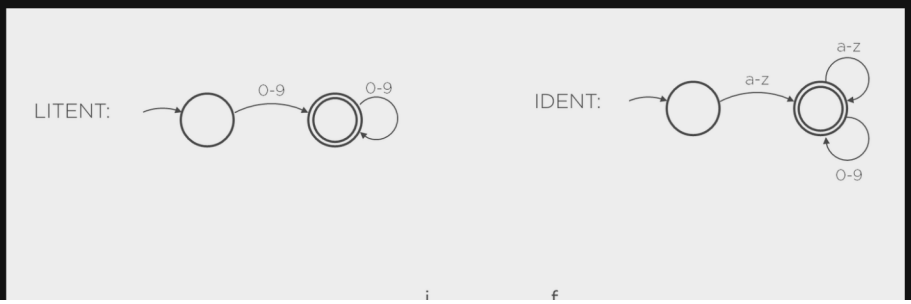
METALENGUAJES

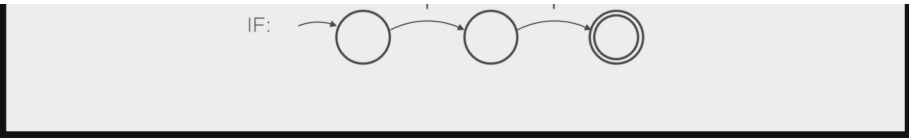
Un metalenguaje deberá de ser preciso y concreto para que no haya problemas de repetición o de confusión es los patrones.

Los dos metalenguajes más usados para reglar estos aspectos son los autómatas finitos y las expresiones regulares.

Autómatas finitos

Habría que crear un autómata finito por cada token del lenguaje:





Expresiones regulares

En nuestro caso es lo que vamos a usar ya que es mucho más cómodo y rápido.

Para crear una especificación habría que crear una expresión regular por cada token del lenguaje:

Token	Patrón
LITENT	*(0|1|2|3|4|5|6|7|8|9)+*
LITREAL	*(0|1|2|3|4|5|6|7|8|9)+.(0|1|2|3|4|5|6|7|8|9)**
WHILE	*while*

TAREAS DE UN LÉXICO

En cada llamada al método nextToken(), un analizador lexico tiene que realizar las siguientes tareas:
- Ignorar los comentarios
- Ignorar los espacios en blanco y saltos de linea
- Comprobar los patrones de la especificacion
- Notificar errores

FORMAS DE IMPLEMENTACIÓN

Esta la implementación manual y la implementación a base de herramientas. En el primer caso tenemos implementaciones con tablas de estados o implementaciones estructuradas.

IMPLEMENTACION CON HERRAMIENTAS - ANTLR

ANTLR ya nos proporciona una clase equivalente al analizador lexico hecho a mano. De hecho, incluye:
- Constantes enteras para los tokens
- El método nextToken con la implementación de los patrones (por el método de tabla de estados)

Para probar dicha clase:

```
import org.antlr.v4.runtime.*;


public class Main {

    public static void main(String[] args) throws Exception {

        var lexer = new Lexicon(CharStreams.fromFileName("source.txt"));


        Token token;
        while ((token = lexer.nextToken()).getType() ≠ Lexicon.EOF) {
              System.out.println("Token: " + token.getType());
```

```
                System.out.println("Lexema: " + token.getText());
                System.out.println("Linea: " + token.getLine());
                System.out.println("Columna: " + token.getCharPositionInLine());
            }

            System.out.println("Traza lexer finalizada");
        }
}
```

Operadores de ANTLR

Operador	Descripción	Ejemplo	Lexemas válidos
?	El operando al que se aplique será opcional	*pa?b*	*pb y pab*
.	Representa a cualquier carácter (incluido salto de línea)	*a.b*	*aCb, a$b, a b, ...*
[conjunto]	Representa a *uno* de los caracteres en dicho conjunto. Se permiten rangos de caracteres	*[a-c4]*	*a, b, c y 4*
~	Un carácter que no sea el operando	*p~a0*	*pb0, p$0, p60, p 0, ...*
+?	Versión *non-greedy* del '+'		
*?	Versión *non-greedy* del '*'		

En el caso de los operadores non-greedy lo que sucede es que cuando se encuentra un lexema válido se deja de reconocer. Por ejemplo:
- T: '@' .* '@';
    - Reconoce @hola@@adios@
- T: '@' .*? '@';
    - Reconoce @hola@

Prioridades de las Reglas

Norma 1: Cuando una entrada puede ser reconocida por más de una regla, se elige aquella que forme el lexema más largo.

Norma 2: Cuando varias reglas forman un lexema del mismo tamaño, se elige la regla que se haya definido primero