PROCESADORES DE LENGUAJE

Redactado por: Miguel Herrero Obeso
Actualizado el: 11/06/2005
Impartido por: Raúl Izquierdo

Nota: Estos apuntes pretenden ser un reflejo de lo explicado en clase y están pensados para compaginarlos con las transparencias/apuntes de la asignatura. No me responsabilizo de los suspensos que originen estos apuntes. Si consideras que alguna parte debería ser corregida, házmelo saber enviándome un correo-e a través de mi página web.

1 Índice

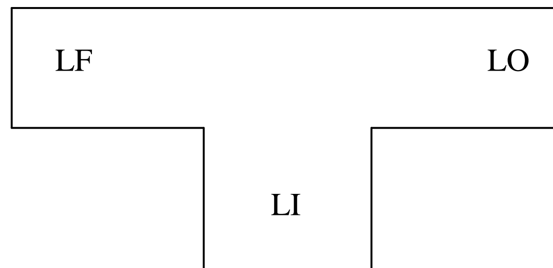
1 Índice ... 1
2 Conceptos básicos ... 1
3 Análisis léxico ... 7

2 Conceptos básicos

2.1 Definiciones

Un **procesador** de lenguaje es el nombre genérico que reciben las aplicaciones cuya entrada es un lenguaje (entrada no trivial).

Un **traductor** es un procesador de lenguaje cuya salida también es un lenguaje. En un traductor intervienen tres lenguajes:



LF: lenguaje fuente, o de entrada (por ejemplo, java).
LO: lenguaje objeto, o de salida (por ejemplo, bytecode).
LI: lenguaje de implementación (por ejemplo, C).

Un **compilador** es un traductor cuyo lenguaje de entrada es de alto nivel y el de salida es de bajo nivel.

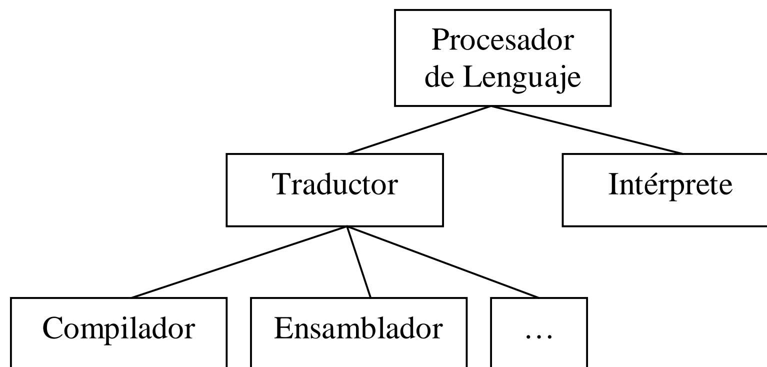
Un **compilador** cruzado es aquel compilador cuyo código de destino es para una máquina distinta de la que ejecuta el compilador.

Un **ensamblador** es un traductor cuyo lenguaje de entrada es ensamblador y la salida es código máquina.

Un **macro-ensamblador** es un ensamblador avanzado con instrucciones más complejas, similares a C.

Un **decompilador** es aquél compilador cuyo lenguaje de entrada es de bajo nivel y el de salida es de alto nivel.

Un **desensamblador** es un traductor cuyo lenguaje de entrada es código máquina y el lenguaje de salida es ensamblador.



2.2 Lenguajes y gramáticas

Una **gramática** describe un lenguaje. Dada una sentencia y usando la gramática, podemos saber si pertenece al lenguaje. Una gramática está formada por:

G={V_T,V_N,S,P}

Vocabulario Terminal (V_T): símbolos que forman las sentencias del lenguaje.
Vocabulario no Terminal (V_N): símbolos auxiliares que se usan en las producciones. No pueden aparecer en las sentencias del lenguaje.
Símbolo inicial (S): símbolo del V_N a partir del cual obtendremos todas las sentencias del lenguaje.
Conjunto de producciones (P): transformaciones que convierten símbolos en un resultado, por el cual se pueden cambiar las sentencias.

Un lenguaje es el conjunto de todas las sentencias que pueden generarse a partir de una gramática. Se dice que una sentencia pertenece al lenguaje si:

1. Está formada por el primer símbolo de V_T.
2. Partiendo del símbolo inicial S y aplicando transformaciones podemos llegar a dicha sentencia.

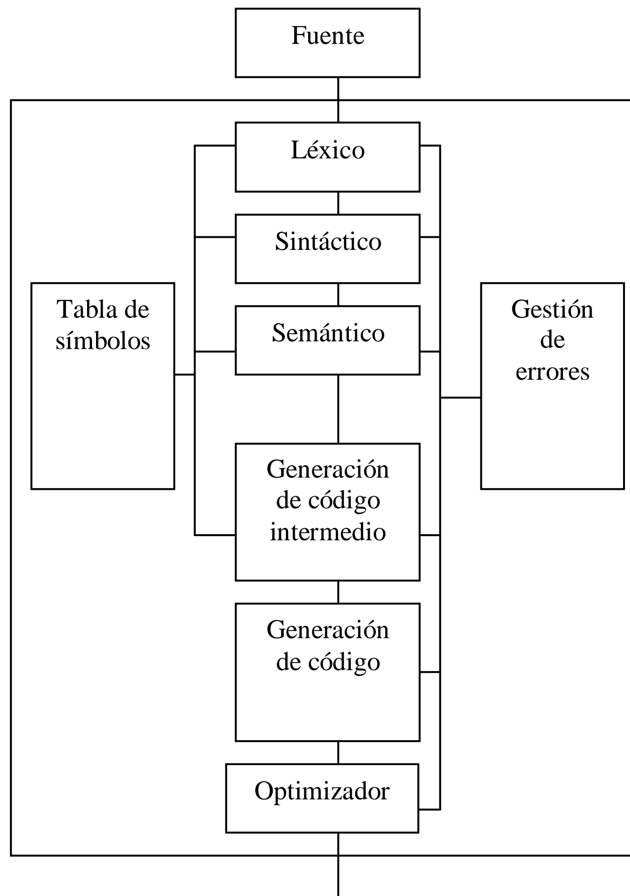
```
Ejemplo:

G = ({a,b},{S,A},S,P)
```

```
P = S → Aa, A → bA | λ

bba pertenece al lenguaje porque S→Aa→bAa→bbAa→bba
```

2.3 Estructura general de un traductor



Se suele dividir en dos fases:

1. **Análisis** (léxico, sintáctico y semántico): analiza la fuente para garantizar la pertenencia al lenguaje.
2. **Síntesis** (generación de código intermedio, generación de código y optimizador).

A continuación vamos a realizar una pequeña introducción de cada módulo:

Análisis léxico

1. Lee caracteres de uno en uno.
2. Agrupa caracteres.
3. Los valida.
4. Clasifica los válidos en grupos.

Análisis sintáctico

La sintaxis es el conjunto de reglas que especifica la formación de los programas. El sintáctico nos dice, de los lexemas válidos, cuáles están ordenados correctamente.

Análisis semántico

Se realizan comprobaciones muy específicas al lenguaje, por ejemplo:

- Una variable ha sido definida solo una vez
- Al llamar a una función, los parámetros tienen que tener un tipo.
- Una variable tiene que estar inicializada.

Gestión de errores
Módulo de apoyo para centralizar la gestión de errores.

Tabla de símbolos
Almacena elementos (variables, definición de funciones, …). Es una estructura de datos que contiene toda la información relativa a cada identificador. Cada símbolo tiene dos partes: el nombre y una serie de atributos. Se suelen implementar sobre tablas hash, porque hay muchísimas más búsquedas e inserciones que borrados.

Generación de código intermedio
Para evitar crear compiladores a medida, se genera código intermedio, de forma que ese código pueda ser transformado posteriormente a, por ejemplo, Linux o Windows.

Generación de código
Crea las instrucciones reales para el procesador. Es específico de plataforma.

Optimización
Manteniendo la funcionalidad, debe hacer el código más eficaz. Se puede dividir en dos optimizaciones (una antes de la generación de código y otra después).

La primera optimización es independiente de plataforma. Por ejemplo:

```
a = 5 + MAX → a = 105
```

La segunda optimización realiza modificaciones dependientes de plataforma.

Todas estas fases anteriores se pueden dividir de otra manera:

- **Frontend**: aquellas fases que dependen del lenguaje de entrada (léxico, sintáctico, semántico y generación de código intermedio).
- **Backend**: aquellas dependientes del lenguaje de salida (generación de código y optimización).

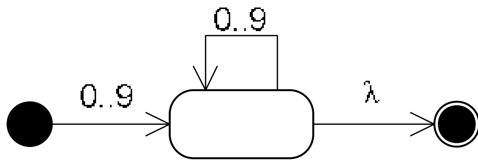
2.4 Metalenguajes

Son lenguajes no ambiguos que facilitan la implementación. Hay metalenguajes específicos para cada fase.

2.4.1 Metalenguajes léxicos

2.4.1.1 *Autómatas finitos*

Por cada palabra, creamos un autómata.



Con este autómata representamos números reales.

Esta solución es simple pero no es práctica, porque no resulta demasiado eficiente y es difícilmente automatizable.

2.4.1.2 *Expresiones regulares*

La ventaja principal es que son texto. Utiliza cinco operadores:

- * *[repetición]*: c* = {λ, c, cc, …}
- ( ) *[agrupar]*
- + *[cierre positivo]*: c+ = {c, cc, …}
- ‘ ‘ *[concatenación]*
- | *[alternativa]*: a|b

2.4.2 Metalenguajes sintácticos

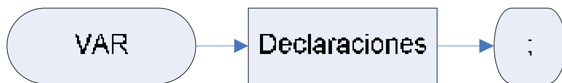
2.4.2.1 *Diagramas sintácticos*

Son representaciones de la sintaxis de un lenguaje con un grafo dirigido en el que los vértices son elementos del V_N y V_T.

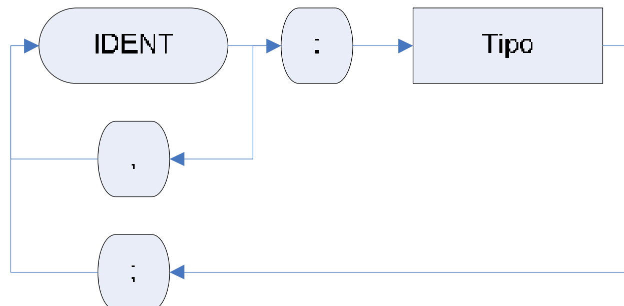
Ejemplo:

Pascal

Variables

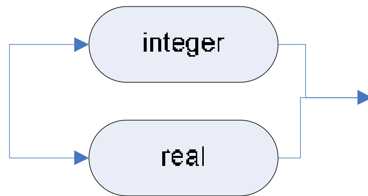


Declaraciones



Bloque

Tipo



2.4.2.2 *BNF*

La notación original es similar a esta → <bloque>::=begin<instrucciones>end

Es decir, constaba de los siguientes elementos:

- <no Terminal>
- ::=
- |

En la práctica, las herramientas utilizan la siguiente notación:

```
BEGIN instrucciones END
```

Con el ejemplo anterior de Pascal obtenemos:

```
Pascal → PROGRAM IDENT PTOYCOMA variab. proced. Bloque PUNTO
Identificador → IDENT | identificador COMA IDENT
Declaración → identificador ':' tipo
Declaraciones → Declaración|Declaración PTOYCOMA Declaraciones
```

2.4.2.3 *EBNF*

BNF extendido. Se utilizan las llaves para realizar iteraciones:

{a} = cero o más veces 'a'.

Podemos cambiar los límites superior e inferior:

$\{a\}_3^8$

```
Ejemplo:

Declaraciones → IDENT {COMA IDENT} DOSPUNTOS Tipo {PUNTOYCOMA
IDENT {COMA IDENT} DOSPUNTOS Tipo}
```

2.4.3 Metalenguajes semánticos

2.4.3.1 *Gramáticas atribuidas*

Partimos de una gramática libre de contexto (BNF, por ejemplo). A cada símbolo de la gramática se le puede asociar un conjunto finito de atributos. En cada producción se indica como se manipulan los atributos. Una gramática atribuida describe un sublenguaje mediante las condiciones que deben cumplir los atributos.

```
Ejemplo:

expr → expr MAS expr | número
número → entero | real

Vamos a evitar que se sumen números de distinto tipo.
Asignamos atributos expr.tipo y numero.tipo

expr0 → expr1 MAS expr2
{si expr1.tipo distinto expr2.tipo ERROR expr0.tipo =
expr1.tipo}

numero → entero {numero.tipo = 'E'} | real {numero.tipo = 'R'}
```

2.5 Validación

Para garantizar la validez de nuestro procesador de lenguaje habría que realizar 6 tipos de tests:

1. **Tipo A**: programa correcto, no debe mostrar errores
2. **Tipo B**: programa incorrecto, debe mostrar errores
3. **Tipo L**: programa que fuerza a errores de vinculado
4. **Tipo C**: programa correcto, comprobamos que el código generado es el esperado
5. **Tipo D**: comprueba las capacidades máximas del procesador (recursividad, anidamiento, …)
6. **Tipo E**: comprueba si existen ambigüedades

2.6 Documentación

- **Manual del programador**: define el lenguaje de entrada.
- **Manual de usuario**: indica como se instala, si tiene IDE, etc.
- **Manual de referencia técnica**: manual interno a los desarrolladores que describe los módulos y la comunicación entre ellos.

3 Análisis léxico

3.1 Definiciones

Un **lexema** es una secuencia de caracteres que constituyen un símbolo terminal de la gramática.
Un ***token*** es un conjunto de lexemas que pueden tratarse como una unidad sintáctica.

Un **patrón** es una regla que permite determinar qué lexemas pertenecen a un *token*.

```
Ejemplo:
Token → entero
Patrón → [0-8]+
Lexema → 12
```

3.2 Pasos para construir un analizador léxico

1. Definir los *tokens* del lenguaje
2. Para cada token hay que definir un patrón
3. Implementamos los patrones

3.3 Interfaz del léxico

Los *tokens* se suelen representar con un interfaz de enteros. Por comodidad, el entero correspondiente a un *token* de un solo carácter suele tener el valor ASCII de dicho carácter. Para que otros *tokens* no colisionen con su valor, al resto de *tokens* se les asigna valores mayores que 256.

Es conveniente que al preguntar por el lexema nos devuelva su tipo correspondiente, por lo que encapsularemos dicha funcionalidad en la clase *ParserVal*.

```
class ParserVal
{
public int ival;
public double dval;
public string sval;
public Object obj;
}
```

No todo *token* debe tener lexema asociado (por ejemplo, IF, WHILE, etc.). Por convenio, existe un *token* para indicar el fin de fichero, que se suele representar con el valor 0.

Otra tarea es la eliminación de comentarios, espacios en blanco, tabulaciones, saltos de línea, etc. También hay que llevar la cuenta de la línea en la que estamos, para facilitar la localización de errores.

El interfaz hasta ahora descrito quedaría así:

```
class Yylex
{
Yylex(Reader input){…} // Constructor que lee de un flujo
int yylex(){…} // Devuelve el siguiente token a leer
ParserVal lexeme() {…} // Devuelve el lexema asociado
}
```

Para probar el analizador, podemos utilizar el siguiente código:

```
int main()
```

```
{
Yylex lex = new Yylex(new FileReader(argv[0]));
int token;
while ((token = lex.yylex()) != 0)
     System.out.println(lex.getLine()+token+"="+lex.lexeme());
}
```

3.4 Implementación con herramientas

Nosotros vamos a utilizar una herramienta Java denominada **JFlex**. El flujo de trabajo de la misma sería el siguiente:

Los pasos a seguir para generar el analizador son los siguientes:

1. Escribimos el fichero con las expresiones regulares

```
//lexico.l
%%
% byaccj
% unicote
%%
[0-9]+{System.out.println("numero"+yytext());}
.|\n {}
```

2. Procesamos el fichero con la herramienta

```
Java -jar jflex.jar lexico.l
```

Nos encontraremos con un fichero yylex.java que contiene una clase similar a la que describimos anteriormente.

3.4.1 Formato del fichero de especificación de JFlex

Existen tres secciones separadas por la cadena %%, lo que forma una estructura similar a esta:

```
Sección 1 - código de usuario
%%
Sección 2 - opciones y declaraciones
%%
Sección 3 - reglas
```

3.4.1.1 *Código de usuario*

El código que se encuentre aquí se copia antes de la clase Yylex. Es útil para añadir sentencias *import* o *package*.

3.4.1.2 *Opciones y declaraciones*

Se utiliza para tres cosas: directivas, macros y estados.

Las **directivas** indican como generar código. Se definen con un % al comienzo de la línea. Cada símbolo % debe ocupar una línea distinta. A continuación describimos algunas de ellas:

- *% class <nombre>* - permite cambiar el nombre de la clase generada
- *% function <nombre>* - permite cambiar el nombre del método para leer un token
- *%{ … %}* - permite incluir código java que va dentro de la clase
- *% init{ … %init}* - similar a la anterior pero que incluye el código en el constructor
- *% debug* - dentro de la clase Yylex, añade un *main* que construye un léxico y le pasa *argv*, realizando una traza de la lectura de fichero
- *% byaccj* - genera código compatible con la herramienta que emplearemos para el análisis sintáctico
- *% unicote* - contempla el juego de caracteres unicote en los ficheros
- *%line* - crea una variable yyline que lleva la cuenta (automáticamente) de la línea actual en la que se está leyendo *tokens*
- *%column* - crea una variable yyline que lleva la cuenta (automáticamente) de la columna actual en la que se está leyendo *tokens*

El resto de directivas conviene leerlas del manual de JFlex.

Las **macros** consisten en la asignación de un nombre a una expresión regular, de forma que es más sencillo referirnos a ella.

```
entero=[0-9]+
…
{entero}{…}
```

Tenemos que poner la macro entre llaves para que la reconozca como macro y no como la cadena 'entero'. Es posible utilizar macros en definiciones de macros:

```
real = {entero}"."{entero}
```

Los **estados** no los veremos en esta asignatura.

3.4.1.3 *Reglas*

Una regla es una pareja formada por un **patrón** y una **acción**. La acción debe ir entre llaves.

- Patrones
  - Operadores
    - a|b - alternativa
    - ab - concatenación
    - a* - cero o más
    - a* - uno o más
    - a? - opción (cero o uno)
    - !a - negación
    - ~a - cualquier secuencia que termine en esto
    - (a) - agrupar
    - Precedencia (de mayor a menor)
      - Operadores unarios postfijos
      - Operadores unarios prefijos
      - Concatenación
      - Alternativa
  - Secuencias de escape
    - \n, \t, \r, *, … - secuencias de escape
    - Entre comillas, todos los operadores pierden su valor menos las secuencias de escape y el guión
  - Conjuntos
- Acciones
  - String yytex() – la secuencia de caracteres que ha cumplido el patrón (lexema)
  - int yylength() – longitud del lexema
  - char yycharat(int index) – carácter del lexema en una posición determinada
  - int yyline – número de línea actual
  - int yycolumn – número de columna actual

```
Ejemplo:

ab|cd+ es equivalente a (ab)|(c(d+)) aplicando precedencia de
operadores.
```

```
Ejemplo:

Vamos a normalizar nombres utilizando un léxico.

Queremos convertir nombres del estilo de
Friends.-.3x01.El.del.pato-[DVDRip by …](Dual…).avi
en un nombre normalizado como
Friends 3x04 El del pato.avi

Vamos a definir el siguiente analizador léxico:

%%
% byaccj
%{
    private String name=" ";
    public String getName(){
          return name;
    }
```

```
%}
%%
"avi"|"mpg" {name = name.trim()+"."+yytext();}
[.\- ]+ {name += " ";} → cambia '.', '-' o ' ' por espacios
("["~"]"|"("~")")+ {} → elimina los corchetes
. {name += yycharat(0);} → copia los caracteres restantes
```

3.4.1.4 *Resolución de conflictos*

```
[0-8]+ {System.out.println(1);}
[0-9]+ {System.out.println(2);}
```

Si suponemos que hemos programado el código anterior y la entrada es 32951 ¿qué patrones resultan activados?

Cuando la entrada casa con varios patrones, se elige aquél que forma el lexema más largo. En el ejemplo anterior se elegiría el segundo patrón. En el caso de que coincida la longitud, se elige el que primero se haya definido en el fichero.

Cuando la entrada no casa con ningún patrón, se lanza una excepción. Para evitar esto es recomendable poner una regla que se salte los caracteres sobrantes, como la siguiente:

```
. {}
```

```
Ejercicio:

Implementar un analizador léxico que lea identificadores,
constantes (números), el operador '=' y el ';'. Además tiene
que eliminar espacios, comentarios y si un carácter no
pertenece a un identificador o constante, debe mostrar un error
por pantalla.

%%
% byaccj
% unicode
% line
%{
    private ParserVal lexeme;
    public ParserVal lexeme() {return lexeme;}
    public int line() {return line;}
%}
%%
"=" {return '=';}
";" {return ';';}
[a-zA-Z][a-zA-Z0-9]* {lexeme = new ParserVal(yytext()); return
tokens.IDENT}

[ \t\n\r] {}
[0-9]+ {lexeme = new ParserVal(Integer.ParseInt(yylex()));
return tokens.CONST;}
```

```
. {System.out.println("Error");
```

Si quisieramos realizar el analizador nosotros mismos, la clase resultante sería similar a esta:

```
class yylex{
    private int currentChar;
    private char readNext(){
        currentChar = input.read();
        return (char)currentChar;
    }
    private char getChar(){return (char)currentChar;}
    private boolean noMoreChars(){return (currentChar == -1);}
    private Reader input;

    public yylex(Reader reader){
        input = reader;
        readNext();
    }
    private ParserVal lexeme;
    public ParserVal lexeme(){ return lexeme;}
    int yylex(){
        while(true){
            while(getChar()==" "||getChar()=='\n')
                readNext();
            if (noMoreChars()) return 0;
            if (getChar()==';'){
                readNext();
                return ';';
            }
            if (Character.isDigit(getChar()){
                StringBuffer buffer = new StringBuffer();
                buffer.append(getChar());
                while(Character.isDigit(readNext())
                    buffer.append(getChar());
                lexeme = new ParserVal(Integer.parseInt
                                (buffer.toString);
                return tokens.CTE;
            }
        }
    }
```