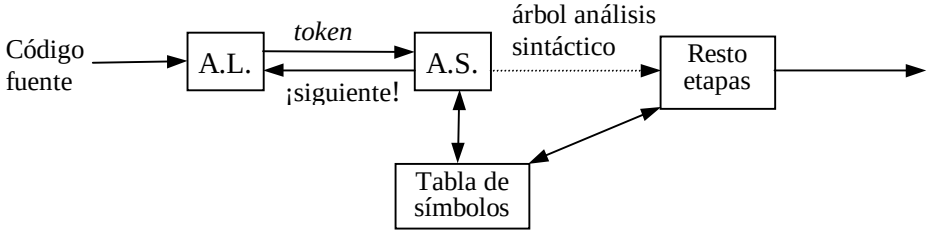
TEMA 3

ANÁLISIS SINTÁCTICO

3.1 LA FUNCIÓN DEL ANALIZADOR SINTÁCTICO

Como ya se indicó en la introducción, la principal tarea del analizador sintáctico (o *parser*) no es comprobar que la sintaxis del programa fuente sea correcta, sino construir una representación interna de ese programa y, en el caso en que sea un programa incorrecto, dar un mensaje de error. Para ello, el analizador sintáctico (A.S.) comprueba que el orden en que el analizador léxico le va entregando los *tokens* es válido. Si esto es así significará que la sucesión de símbolos que representan dichos *tokens* puede ser generada por la gramática correspondiente al lenguaje del código fuente.

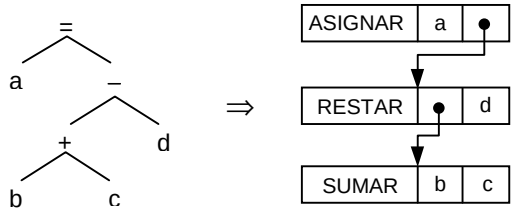


La forma más habitual de representar la sintaxis de un programa es el árbol de análisis sintáctico, y lo que hacen los analizadores sintácticos es construir una derivación por la izquierda o por la derecha del programa fuente, que en realidad son dos recorridos determinados del árbol de análisis sintáctico. A partir de ese recorrido el analizador sintáctico debe construir una representación intermedia de ese programa fuente: un árbol sintáctico abstracto o bien un programa en un lenguaje intermedio; por este motivo, es muy importante que la gramática esté bien diseñada, e incluso es frecuente rediseñar la gramática original para facilitar la tarea de obtener la representación intermedia mediante un analizador sintáctico concreto.

EJEMPLO:

Los *árboles sintácticos abstractos* son materializaciones de los árboles de análisis sintáctico en los que se implementan los nodos de éstos, siendo el nodo padre el operador involucrado en cada instrucción y los hijos sus operandos. Por otra parte, las representaciones intermedias son lenguajes en los que se han eliminado los conceptos de mayor abstracción de los lenguajes de programación de alto nivel. Sea la instrucción "`a=b+c–d`". A continuación se muestra su representación mediante ambos esquemas citados.

Árbol sintáctico abstracto	Lenguaje intermedio

	`sumar b c t1`
	`restar t1 d t2`
	`asignar t2 a`

❑

El A.S. constituye el esqueleto principal del compilador. Habitualmente el analizador léxico se implementa como una rutina dentro del sintáctico, al que devuelve el siguiente *token* que encuentre en el *buffer* de entrada cada vez que éste se lo pide. Así mismo, gran parte del resto de etapas de un programa traductor están integradas de una u otra forma en el analizador sintáctico.

Principalmente hay dos opciones para implementar un *parser*:

1. "*a mano*", utilizando una serie de técnicas que se describirán en los siguientes temas;
2. utilizando un generador de analizadores sintácticos (p.ej. el YACC).

Como siempre, ambos enfoques tienen ventajas e inconvenientes, muy similares al caso de los analizadores léxicos (para el segundo caso el inconveniente de la ineficiencia y la ventaja de la sencillez, y viceversa para el primero). En este curso nos centraremos en las técnicas para implementar a mano *parsers* para determinados tipos de gramáticas.

3.2 NOTACIÓN EBNF

EBNF son las siglas de *Extended Backus-Naur Form*. La idea surgió como herramienta para reducir el número de producciones en las gramáticas. Para ello se añaden unas notaciones adicionales a las ya contenidas en la notación BNF.

1.- *Alternativas de una regla:* Como un mismo símbolo auxiliar puede definirse como varios símbolos, utilizamos una única producción utilizando el símbolo '|' para separar las distintas posibilidades que definen al no terminal de la izquierda.

Ejemplo: si $A \rightarrow a$
$A \rightarrow b$
$A \rightarrow c$
estas tres producciones las resumimos en una equivalente: $A \rightarrow a \mid b \mid c$

Otro: <entero> → <digito> | <entero> <dígito>

2.- *Llaves:* { }, lo que aparece entre llaves se repite de cero a *n* veces.

Ejemplos: <Lista_parámetros> → <parámetro> { , <parámetro> }

<tren> → <locomotora> { <vagón> }

3.- *Llaves con repetición especificada:* $\{ \}_x^y$, lo que aparece entre llaves se repite un número de veces comprendido entre *x* e *y*.

Ejemplo: tren con 3, 4 ó 5 vagones: <tren> → <locomotora> $\{ \text{<vagón>} \}_3^5$

4.- *Corchetes:* [ ], lo que está entre los corchetes puede o no aparecer. Es un caso particular de 3.-, pues es equivalente a $\{ \}_0^1$

Ejemplo: Sentencia IF completa en Pascal:

IF <expresión> THEN <sentencia> [ELSE <sentencia>]

3.3 DISEÑO DE GRAMÁTICAS PARA LENGU AJES DE PROGRAMACIÓN

El diseño de gramáticas para lenguajes de programación es una materia que difícilmente puede enseñarse en su totalidad, sino que debe ser aprendida en la mayor medida posible. Sin embargo, la forma de recoger parte de la semántica de los operadores en la gramática es bastante sencilla de explicar. A continuación vamos a ver cómo se plasma en el aspecto de las reglas sintácticas algunas propiedades de los operadores y operandos en los lenguajes de programación.

3.3.1 RECURSIVIDAD

Una de las principales dificultades a la hora de diseñar un compilador es que debe procesar correctamente un número, en principio, infinito de programas distintos. Por otro lado, es evidente que la especificación sintáctica de un lenguaje debe ser finita. El concepto que hace compatible las dos afirmaciones anteriores es el de recursividad. Ésta nos permite definir sentencias complicadas con un número pequeño de sencillas reglas de producción.

EJEMPLO:

Supongamos que queremos expresar la estructura de un tren formado por una locomotora y un número cualquiera de vagones detrás. Si lo hicieramos de esta forma:

tren → locomotora
tren → locomotora vagón
tren → locomotora vagón vagón
. . .

necesitaríamos infinitas reglas de derivación (una por cada número de vagones posibles en el tren). Para expresar lo mismo con un par de sentencias podemos utilizar la recursividad de la siguiente manera:

1º) definimos la regla base (no recursiva), la cual define el concepto elemental de partida y que en este caso sería:

tren → locomotora

2º) definimos una o más reglas recursivas que permitan el crecimiento ilimitado de la estructura partiendo del concepto elemental anterior. En este caso una nos basta:

tren → tren vagón

y con esto nos ahorramos la utilización de muchas más reglas.

❑

Estructura de la recursividad:

1. Regla no recursiva que se define como caso base.
2. Una o más reglas recursivas que permiten el crecimiento a partir del caso base.

EJEMPLO:

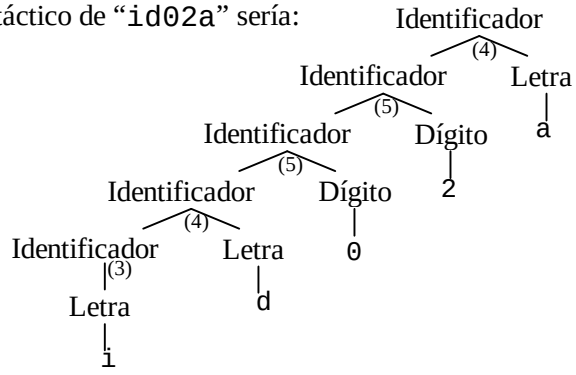
La gramática para describir un identificador en la mayoría de los lenguajes de programación, escrita como una gramática independiente del contexto, podría ser como la siguiente:

N = { Letra, Dígito, Identificador}
T = { a, b, c,..., z, 0, 1,..., 9 }
P = { Letra → a
Letra → b
...
Letra → z
Dígito → 0
Dígito → 1
...
Dígito → 9
Identificador → Letra (3)
Identificador → Identificador Letra (4)
Identificador → Identificador Dígito (5) }
S = Identificador

(1)
(2)

Notación:
N = conjunto de no terminales
T = conjunto de terminales
P = conjunto de producciones
S = Símbolo inicial o axioma

Así, el árbol sintáctico de "`id02a`" sería:



❑

Definición:

Una gramática se dice que es recursiva, si podemos hacer una derivación (sucesión de una o más producciones) de un símbolo no terminal tras la cual nos vuelve a aparecer dicho símbolo entre los símbolos de la parte derecha de la derivación. $A \Rightarrow \alpha A \beta$

Unos casos especiales de recursividades son aquellos en los que aparecen derivaciones como $A \Rightarrow A\beta$ o $A \Rightarrow \alpha A$ y se denominan *recursividad izquierda* y *derecha*, respectivamente.

Un no terminal A se dice que es recursivo si a partir de A se puede derivar una forma sentencial en que aparece él mismo en la parte derecha.

3.3.2 AMBIGÜEDAD

Una gramática es **ambigua** si el lenguaje que define contiene alguna sentencia que tenga más de un único árbol de análisis sintáctico. Es decir, si se puede construir más de un árbol de análisis sintáctico quiere decir que esa sentencia puede "querer decir" cosas diferentes (tiene más de una interpretación). Una gramática es **no ambigua** cuando cualquier tira del lenguaje que representa, tiene un único árbol sintáctico.

Como veremos más adelante, no es posible construir analizadores sintácticos eficientes para gramáticas ambiguas y, lo que es peor, al poderse obtener más de un árbol sintáctico para la misma cadena de entrada, es complicado conseguir en todos los casos la misma representación intermedia. Por estos motivos debemos evitar diseñar gramáticas ambiguas para los lenguajes de programación; aunque no disponemos de técnicas para saber a priori si una gramática es ambigua o no, si ponemos ciertas restricciones a la gramática estaremos en condiciones de afirmar que no es ambigua.

La única forma de saber que una gramática es ambigua es encontrando una cadena con dos o más árboles sintácticos distintos (o dos derivaciones por la izquierda). Las gramáticas que vamos a utilizar normalmente generan lenguajes infinitos, por tanto no es posible encontrar en un tiempo finito esa cadena con dos o más árboles sintácticos. Sin embargo, si la gramática tiene alguna de las siguientes características, es sencillo encontrar una cadena con dos o más árboles:

- Gramáticas con ciclos simples o menos simples:

  S → A
  S → a
  A → S

- Alguna regla con una forma

E → E **...** E

con cualquier cadena de terminales y no terminales entre las dos *E*. Es posible que con algún terminal antes de la primera *E* o algún terminal después de la última *E* pueda producirse también ambigüedad; por ejemplo, el `if-then-else` de Pascal y C es fácil que se exprese con una construcción ambigua.

- Un conjunto de reglas de forma parecida a:

S → A
S → B
A → B

- Producciones recursivas en las que las variables no recursivas de la producción puedan derivar a la cadena vacía:

S → H R S
S → s
H → h | ε
R → r | ε

- Variables que puedan derivar a la cadena vacía y a la misma cadena de terminales, y que aparezcan juntas en la parte derecha de una regla o en alguna forma sentencial:

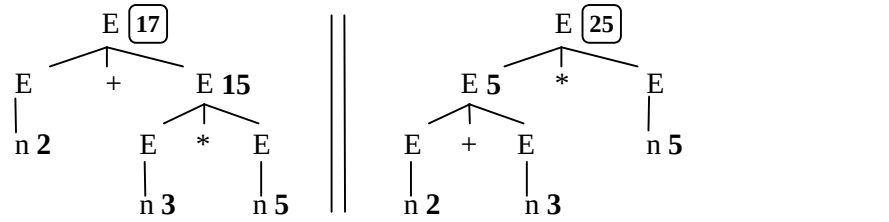
S → H R
H → h | ε
R → r | h | ε

EJEMPLO:

Sea una gramática cuyas reglas de producción son:

E → E + E | E ∗ E | (E) | número

Con estas reglas se puede generar la tira '`2+3*5`' que tiene dos posibles árboles sintácticos. En función de que se escoja uno u otro, el resultado de evaluar dicha expresión matemática es uno u otro, como se ve a continuación:

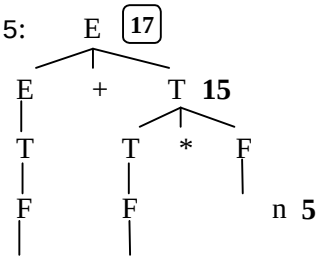


E → E+E → n+E → n+E∗E → n+n∗E → n+n∗n E → E∗E → E+E∗E → n+E∗E → n+n∗E → n+n∗n

Para solucionar esta ambigüedad se deben modificar las reglas de producción de la gramática. En este caso se trata de distinguir en estas expresiones matemáticas lo que es un factor y lo que es un término (producto de dos factores - monomio). Así se establece la jerarquía de precedencias de los operadores:

<expresión> → <expresión> + <término> | <término>
<término> → <término> ∗ <factor> | <factor>
<factor> → (<expresión>) | número

De esta forma sólo hay un posible árbol sintáctico para `2+3*5`:



n **2** n **3**

❑

3.3.3 ASOCIATIVIDAD Y PRECEDENCIA DE LOS OPERADORES

asociatividad

La asociatividad de un operador binario define cómo se operan tres o más operandos; cuando se dice que la asociatividad de un operador '#' es por la izquierda se quiere decir que si aparecen tres o más operandos (con el operador '#' entre cada dos operandos, por supuesto) se evalúan de izquierda a derecha: primero se evalúan los dos operandos de más a la izquierda, y el resultado de esa operación se opera con el siguiente operando por la izquierda, y así sucesivamente.

Si la asociatividad del operador es por la derecha, los operandos se evalúan de derecha a izquierda. En los lenguajes de programación imperativos más utilizados (Pascal, C, C++, Java, etc.) la asociatividad de la mayoría de los operadores y en particular la de los operadores aritméticos es por la izquierda. Por el contrario, en el lenguaje APL, que es un lenguaje orientado al cálculo numérico, la asociatividad de todos los operadores es por la derecha.

EJEMPLO:

Si la asociatividad del operador '#' es por la izquierda, la expresión '`2#a#7.5`' se evalúa operando primero el '`2`' con la variable '`a`', y operando después el resultado con '`7.5`'. Si la asociatividad fuera por la derecha, primero se operarían la variable '`a`' con el '`7.5`', y después se operaría el '`2`' con el resultado de esa operación. La posición de los operandos con respecto al operador suele ser importante, ya que aunque algunos operadores son conmutativos, la mayoría no lo son.

❑

EJEMPLO:

Existen lenguajes que combinan operadores asociativos por la izquierda con otros asociativos por la derecha: en Fortran existen cinco operadores aritméticos: suma ("+"), resta ("–"), multiplicación ("∗"), división ("/") y exponenciación ("∗∗"). Los cuatro primeros son asociativos por la izquierda, mientras que el último lo es por la derecha. Así, tendremos las siguientes equivalencias:

- `A/B/C` se evalúa como `A/B` y el resultado se divide por `C`
- `X**Y**Z` se evalúa como `Y**Z` y `X` se eleva al resultado

❑

La forma de reflejar la asociatividad de un operador en la gramática es la siguiente: cuando la asociatividad del operador es por la izquierda, la regla sintáctica en la que interviene dicho operador debe ser recursiva por la izquierda, y cuando es por la derecha, la regla en la que interviene debe tener recursión por la derecha. Para comprender estas reglas basta con pensar cómo se desarrollarán los árboles sintácticos con ambos tipos de recursividad y cómo se operará en los nodos del árbol a la subida de un recorrido en profundidad por la izquierda.

precedencia

La precedencia de un operador especifica el orden relativo de cada operador con respecto a los demás operadores; de esta manera, si un operador '#' tiene mayor precedencia que otro operador

‘%’, cuando en una expresión aparezcan los dos operadores, se debe evaluar primero el operador con mayor precedencia.

EJEMPLO:

Con los operadores definidos más arriba, si aparece una expresión como ‘2%3#4’, al tener el operador ‘#’ mayor precedencia, primero se operarían el ‘3’ y el ‘4’, y después se operaría el ‘2’ con el resultado de esa operación.

❑

Siguiendo los criterios aritméticos habituales, en la mayoría de los lenguajes de programación los operadores multiplicativos tienen mayor precedencia que los aditivos, por lo que cuando se mezclan ambos tipos de operaciones en una misma sentencia, se evalúan las multiplicaciones y divisiones de izquierda a derecha antes que las sumas y restas. Existen excepciones: en el lenguaje *Smalltalk* no existe precedencia ni asociatividad, todos los operandos se evalúan de izquierda a derecha sin importar lo que aparezca más a la izquierda de ellos.

La forma de reflejar la precedencia de los operadores aritméticos en una gramática es bastante sencilla. Es necesario utilizar una variable en la gramática por cada operador de distinta precedencia. Cuanto más “cerca” esté la producción de la del símbolo inicial, menor será la precedencia del operador involucrado. La noción de cercanía tiene que ver con el número de producciones que hay que llevar a cabo para llegar hasta esa regla desde el símbolo inicial.

parentización

En la mayoría de los lenguajes de programación se utilizan los paréntesis (que son operadores especiales que siempre tienen la máxima precedencia) para agrupar los operadores según la conveniencia del programador y sortear las precedencias y asociatividades definidas en el lenguaje.

Para incluirlos en la gramática, se añade una variable que produzca expresiones entre paréntesis y los operandos (números, variables, etc.) a la mayor distancia posible del símbolo inicial. En esta producción también se pondrían los operadores unarios a no ser que tengan una precedencia menor (ver más adelante).

EJEMPLO:

Supónganse los operadores +, –, con asociatividad por la izquierda (en 6–3–3, se calcula primero 6–3 y después se le resta el otro 3) y los operadores ∗, / con asociatividad por la derecha. Sean para los dos tipos de operadores la precedencia habitual en los lenguajes de programación (∗ y / tienen más precedencia y, por tanto, se evalúan antes que las sumas y las restas) y los paréntesis tienen la máxima precedencia. La gramática que genera las expresiones con estos operadores y además recoge la asociatividad y la precedencia es la siguiente:

E → E + T
E → E – T
E → T
T → F ∗ T
T → F / T
T → F
F → (E)
F → número

Se dice que esta gramática refleja la precedencia y asociatividad de los operadores porque si construimos el árbol sintáctico para una cadena cualquiera, p.ej. ‘12-4-6/2/2’, veremos que los

operandos están agrupados según su asociativiad y precedencia en las ramas del árbol, lo cual facilitará su evaluación en un recorrido del árbol.

❑

Es importante que la gramática refleje la precedencia y asociatividad de los operadores puesto que en la mayoría de los lenguajes objeto los operadores no tienen precedencia o asociatividad (normalmente porque no pueden aparecer expresiones aritméticas complejas), y por tanto si el árbol sintáctico mantiene correctamente agrupados los operandos de dos en dos será más sencillo evaluar la expresión y traducirla a un lenguaje objeto típico, como puede ser el código máquina de un procesador cualquiera.

La forma de recoger la semántica de los operadores unarios y ternarios depende de cada caso, por lo que es necesario estudiar bien el comportamiento de estos operadores para diseñar una gramática para ellos.

EJEMPLO:
En el caso del operador '`!`' de C (el '`not`' de Pascal) la semántica permite que aparezcan varios operadores seguidos (como por ejemplo '`!!!!0`'). En este caso, habría que añadir una regla como la siguiente a la gramática del ejemplo anterior (dependiendo de la precedencia del operador):

$$F \rightarrow \texttt{!}\ F$$

Sin embargo, el caso del operador de signo, la semántica de muchos lenguajes como C, C++, Pascal, etc. no permite que aparezca más de un signo delante de un término, y además se especifica que el signo afecta a todo el término. En el ejemplo anterior, habría que añadir las siguientes reglas:

$$E \rightarrow + T$$
$$E \rightarrow - T$$

❑

3.4 TIPOS DE ANÁLISIS SINTÁCTICO

Los algoritmos de análisis sintáctico general para gramáticas independientes del contexto tienen un coste temporal del orden de $O(n^3)$; este es un coste demasiado elevado para un compilador, por lo que es necesario buscar subclases de gramáticas que permitan un análisis sintáctico en tiempo lineal.

Desde el punto de vista de la teoría de Análisis Sintáctico, hay dos estrategias para construir el árbol sintáctico:

- **Análisis descendente**: partimos de la raíz del árbol (donde estará situado el axioma o símbolo inicial de la gramática) y se van aplicando reglas por la izquierda de forma que se obtiene una derivación por la izquierda de la cadena de entrada. Para decidir qué regla aplicar, se lee un *token* de la entrada. Recorriendo el árbol de análisis sintáctico resultante, en profundidad de izquierda a derecha, encontraremos en las hojas del árbol los *tokens* que nos devuelve el A.L. en ese mismo orden.
- **Análisis ascendente**: partiendo de la cadena de entrada, se construye el árbol de análisis sintáctico empezando por las hojas (donde están los *tokens*) y se van creando nodos intermedios hasta llegar a la raíz (hasta el símbolo inicial), construyendo así el árbol de abajo a arriba. El recorrido del árbol se hará desde las hojas hasta la raíz. El orden en el que se van encontrando las producciones corresponde a la inversa de una derivación por la derecha.

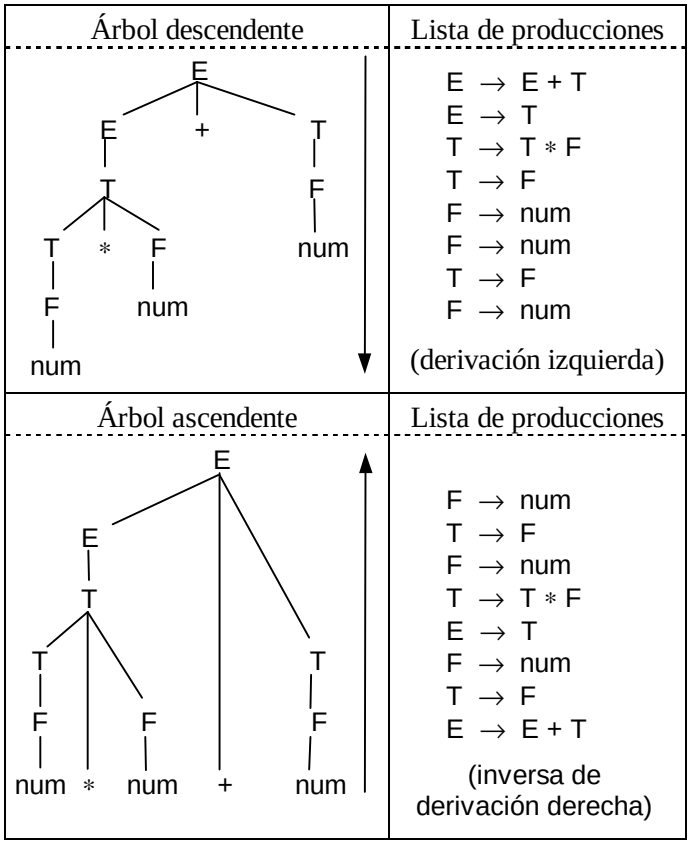
Las dos estrategias recorren la cadena de entrada de izquierda a derecha una sola vez, y necesitan (para que el análisis sea eficiente) que la gramática no sea ambigua.

EJEMPLO:

entrada: "num∗num+num"

Gramática:

E → E + T | T
T → T ∗ F | F
F → num



Obsérvese que en el análisis descendente, partiendo del símbolo inicial hasta alcanzar las hojas, obtenemos una derivación por la izquierda. En el ascendente, partiendo de las hojas hasta llegar al axioma obtenemos la inversa de una derivación por la derecha.

❑

Ambos tipos de análisis son eficientes (coste lineal $O(n)$) pero no son capaces de trabajar con todo tipo de gramáticas (el análisis de tipo general sí que es capaz de tratar cualquier gramática, pero es ineficiente para el diseño de compiladores), pero algunas gramáticas adecuadas para estos tipos de análisis son muy convenientes para describir la mayoría de lenguajes de programación, por ejemplo los siguientes:

- Análisis LL(n)
- Análisis LR(n)

donde:

L → Left to Right: la secuencia de *tokens* de entrada se analiza de izquierda a derecha.
L → Left-most (R = Right-most): utiliza las derivaciones más a la izquierda (a la derecha).
n → es el número de símbolos de entrada que es necesario conocer en cada momento para poder hacer el análisis.

Por ejemplo, una gramática LL(2) es aquella cuyas cadenas son analizadas de izquierda a derecha y para las que es necesario mirar dos *tokens* para saber qué derivación tomar para el no terminal más a la izquierda en el árbol de análisis. Con estos tipos de análisis podemos implementar un analizador para cualquier gramática de contexto libre. Las técnicas para hacer esos diseños las veremos en los próximos temas.

EJERCICIOS

Ejercicio 1: Diseñad una gramática no ambigua para el lenguaje de las expresiones que se pueden construir en *Zaskal* usando únicamente "`true`", "`false`" y operadores booleanos. Los operadores de *Zaskal* son: "`or`" (binario, infijo), "`and`" (binario, infijo), "`not`"(unario, postfijo), "`(`" y "`)`". Sin contar los paréntesis, la precedencia relativa de los operadores es

```
not > and > or
```

además, "`and`" y "`or`" son asociativos por la derecha. Por ejemplo, la expresión

```
(true and false not) or false and true not not
```

sería una expresión correcta en Zaskal (que se evalúa como true, por cierto).

SOLUCIÓN:

E → T or E | T
T → F and T | F
F → F not | (E) | true | false

Ejercicio 2: Diseñad una gramática no ambigua para el lenguaje de las expresiones regulares que se pueden construir con el alfabeto {0,1}. Los operadores que se usan para construir expresiones regulares son los siguientes (ordenados de menor a mayor precedencia):

a|b	unión	binario	asociativo por la izq.
ab	concatenación	binario	asociativo por la dch.
a^+ , a*	clausuras	unarios	

Algunas expresiones regulares que deben poder ser generadas por la gramática son:

```
010
(01*|(0|1+)*|1)1
(0(1+)0|1(0+)1)*0011
(1100*+*)+*
```

Ejercicio 3: Diseñad una gramática no ambigua para los lenguajes que permiten escribir cualquier número de declaraciones de variables enteras, caracteres o reales en Pascal y C.

SOLUCIÓN:

Pascal:

(ejemplo: var a, b, c : char; e, f : integer;)

DecVars → **var** ListDec
ListDec → identificador ListIdent ListDec | identificador ListIdent
ListIdent → dospuntos Tipo ptoycoma | coma identificador ListIdent
Tipo → integer | real | char

C:

(ejemplo: int a, b, c; char e, f ;)

ListDec → Dec ListDec | Dec
Dec → Tipo ListIdent puntocoma
ListIdent → id coma ListIdent | id
Tipo → int | char | float

Ejercicio 4: Diseñad una gramática no ambigua a partir de esta descripción de un lenguaje de programación:

1. Un programa está formado por una secuencia de cero o más declaraciones de variables seguida de una secuencia de una o más instrucciones. Además, cada declaración debe acabar con un punto y coma y entre cada dos instrucciones de la secuencia también debe haber un punto y coma (no debe aparecer punto y coma al final).

2. Una declaración de variables empieza con un tipo, que puede ser entero o booleano (los tipos se representan con las palabras reservadas "`int`" y "`bool`" respectivamente). Después del tipo debe aparecer una secuencia de uno o más identificadores separados por comas. Antes de esta secuencia de identificadores puede aparecer un número entero indicando que los identificadores son *arrays* con ese número de elementos.

3. Una instrucción puede ser cualquiera de las siguientes:

   asignación: está formada por una referencia, un operador de asignación (el símbolo "`:=`") y una expresión. Una referencia es un identificador seguido opcionalmente por una expresión entre corchetes (para indexar los *arrays*);

   entrada: está formada por la palabra reservada "`read`" y una referencia;

   condicional: empieza con la palabra reservada "`if`", una expresión, la palabra reservada "`then`", una secuencia de instrucciones (como se define más arriba) y la palabra reservada "`end`". Opcionalmente, puede aparecer antes de "`end`" la palabra reservada "`else`" y otra secuencia de instrucciones;

   salida: está formada por la palabra reservada "`print`" y una expresión;

   iteración: está formada por la palabra reservada "`while`", una expresión, la palabra reservada "do", una secuencia de instrucciones y la palabra reservada "`end`".

   for: está formada por la palabra reservada "`for`", un identificador, el operador de asignación, una expresión, la palabra reservada "`to`", otra expresión, la palabra reservada "`do`", una secuencia de una o más instrucciones (separadas por un punto y coma entre cada dos instrucciones) y la palabra reservada "`end`".

   selección: La instrucción se define con las siguientes reglas:
   - Empieza por la palabra reservada "`case`", una expresión, una secuencia de uno o más "casos" y la palabra reservada "`end`".
   - Un **caso** empieza por la palabra reservada "`when`", una secuencia de uno o más números enteros sin signo o "rangos" separados por comas, seguida de la palabra reservada "`then`", y una secuencia de instrucciones.
   - Puede aparecer un **caso especial** (que puede ser también el único que aparezca), y debe ser único (no puede haber dos casos especiales) y el último caso. Está formado por la palabra reservada "`else`" y una secuencia de instrucciones.
   - Cuando la expresión no coincida con ninguno de los casos normales, se ejecutará la secuencia de instrucciones asociadas al caso especial si existe.
   - Un **rango** está formado por un número entero, la palabra reservada "`to`" y otro número entero.

4. Las expresiones están formadas por números enteros, las constantes booleanas "`true`" y "`false`", referencias (tal como se definen más arriba), y los operadores "`or`" y "`and`", los operadores relacionales "`!=`", "`==`", "`>=`", "`>`", "`<=`" y "`<`", los operadores de suma ("`+`") y resta ("–"), los operadores de producto ("`*`") y división ("`/`") y el operador "`not`". La

siguiente tabla muestra los operadores ordenados de menor a mayor precedencia y también muestra su asociatividad y aridad:

Operadores	Asociatividad	Orden
`or`	izquierda	binario
`and`	izquierda	binario
`!=, ==, >=, >, <=, <`	izquierda	binarios
`+, –`	izquierda	binarios
`+, –`	-	unarios
`*, /`	izquierda	binarios
`not`	-	unarios

Los operadores "+" y "–" unarios sólo pueden aparecer al principio de una expresión, y sólo afectan al primer término de la expresión. Además, se pueden utilizar paréntesis para agrupar operaciones. En general, las expresiones son similares (en cuanto a su sintaxis) a las de Pascal o C.

Ejercicio 5: Las reglas siguientes definen el lenguaje **LogPro**. Escribid las expresiones regulares que definen los *tokens* de este lenguaje y después diseñad una gramática.

- Un programa en el lenguaje **LogPro** consta de una secuencia de cero o más hechos o reglas y **una** única pregunta.
- Un hecho es un predicado seguido de un punto.
- Una regla es un predicado seguido del símbolo "<-", la parte derecha de la regla y un punto.
- Una pregunta empieza con el símbolo "<-" seguido de la parte derecha de una regla y termina en un punto.
- Un predicado tiene un nombre (que es una secuencia de letras, dígitos y el símbolo "_" (el carácter de subrayado) que empieza por una letra minúscula) y cero o más argumentos separados por comas y encerrados entre paréntesis (al contrario que en C, si no tiene argumentos, no se ponen los paréntesis).
- Un argumento puede ser un número (una secuencia de uno más dígitos sin signo), una variable (una secuencia de letras, dígitos y el símbolo "_", que empieza por una letra mayúscula o por "_") o un predicado.
- La parte derecha de una regla es una expresión booleana, y está formada por una secuencia de términos booleanos combinados con los operadores "," (una coma, representa la operación lógica *and*) y ";" (un punto y coma, representa al *or* lógico). Los dos operandos tienen la asociatividad por la izquierda, y el ";" tiene menor precedencia (se evalúa después) que el ",". Se pueden utilizar paréntesis para agrupar expresiones, de la misma manera que se utilizan en expresiones de C o Pascal.
- Un término booleano puede ser un predicado o una expresión relacional. Una expresión relacional está formada por dos expresiones aritméticas separadas por un operador relacional, que puede ser uno de estos símbolos: "`!=`", "`==`", "`>=`", "`>`", "`<=`" y "`<`".
- Una expresión aritmética está formada por números, variables, paréntesis y los operadores "+", "–", "*" y "/", con la misma precedencia y asociatividad que tienen en lenguajes como C o Pascal (una expresión aritmética en LogPro es sintácticamente correcta en C o Pascal).

- De igual forma que en C o Pascal, se pueden utilizar espacios en blanco, tabuladores y cambios de linea para mejorar la legibilidad de un programa en LogPro, pero no son elementos del lenguaje.
- Un argumento de un predicado puede ser también una lista, que está formada por un corchete izquierdo, una secuencia de elementos separados por comas, una cola opcional y un corchete derecho. Cualquier tipo de argumento de un predicado puede ser un elemento de una lista (se puede construir una lista de listas), y la cola (que puede aparecer o no) está formada por el símbolo "|" seguido de una lista. La cola, si aparece, va situada inmediatamente antes del corchete derecho (aunque puede haber blancos entre estos elementos).

EJEMPLO:

```
preD1   .
preD2(23,_23,f(a)) <- (eurt,eslaf;cierto), 2+3  *4<= 3, X == 5.
preD3(Predicado, [l,i,s,t,a
       |[c,o,l,a]]).
<-preD4(preD2(Y,2,f(X)),preD1).
```

Ejercicio 6: Diseñad una gramática *no ambigua* que genere el lenguaje G definido por las siguientes frases:

- Un programa en G es una secuencia de uno o más métodos.
- Un método está formado por una declaración de variables y una secuencia de cero o más mensajes acabada en una sentencia de retorno.
- Una declaración de variables está formada por una barra vertical ("|"), una secuencia de cero o más identificadores y otra barra vertical.
- Un mensaje puede ser una asignación, una expresión o un mensaje de método.
- Un mensaje de asignación está formado por una variable, el simbolo ":=" y un mensaje.
- Las expresiones pueden contener números enteros, variables, los operadores "+", "–", "∗" y "/", con la misma asociatividad que en C o Pascal, pero *sin precedencia*. Una expresión puede ser un mensaje entre paréntesis.
- Un mensaje de método está formado por un identificador, un corchete izquierdo, una secuencia de cero o más expresiones separadas por un punto y coma entre cada dos expresiones, y un corchete derecho.
- Una sentencia de retorno está formada por el símbolo "^".

Ejercicio 7: Diseña una gramática *no ambigua* que genere (carácter a carácter) el lenguaje de todos los números enteros sin signo que tengan un número par de cifras, considerando que el 0 no es par. Algunos números que tendría que generar esta gramática son:

00
0135
011112
2358
00110073
