Tema 3

Introducción al análisis sintáctico

Bibliografía:

- Aho, A.V., Sethi, R., Ullman, J.D. (1990), *Compiladores: principios, técnicas y herramientas*, Tema 4, pág.: 163-186.
- Louden, K.C. (1997), *Compiler Construction: Principles and Practice*, Tema 3, páginas: 93-140.

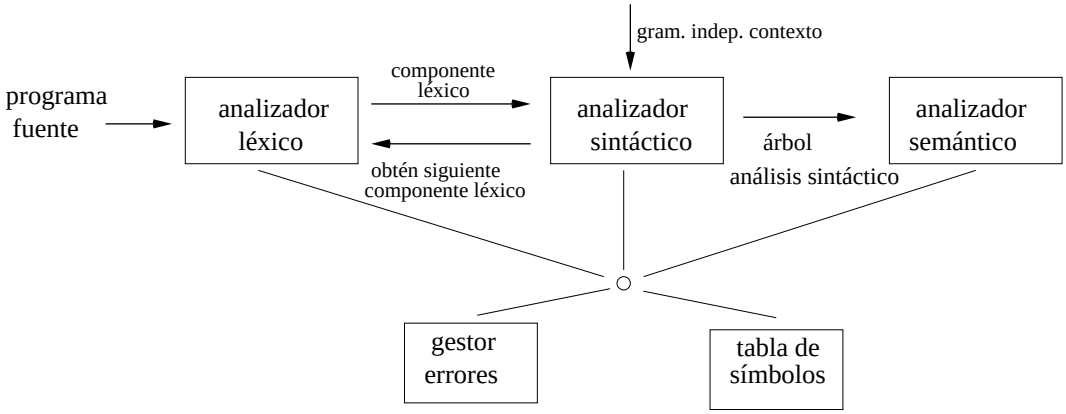
Contenidos:

1. El proceso de análisis sintáctico.
2. Especificación sintáctica de los lenguajes de programación:
   a) Gramáticas independientes del contexto versus expresiones regulares.
   b) Gramáticas independientes del contexto versus gramáticas dependientes del contexto.
3. Derivaciones y árboles sintácticos.
4. Gramáticas limpias y bien formadas.
5. El problema de la ambigüedad en las gramáticas. Eliminación:
   a) Mediante reglas de precedencia.
   b) Mediante transformación.
6. Clasificación de los métodos de análisis sintáctico.

3.1. El proceso de análisis sintáctico

Funciones del analizador sintáctico:

- Comprobar si la cadena de componentes léxicos proporcionada por el analizador léxico puede ser generada por la gramática que define el lenguaje fuente (Gramática Independiente del Contexto, GIC).
- Construir el árbol de análisis sintáctico que define la estructura jerárquica de un programa y obtener la serie de derivaciones para generar la cadena de componentes léxicos. El árbol sintáctico se utilizará como representación intermedia en la generación de código.
- Informar de los errores sintácticos de forma precisa y significativa y deberá estar dotado de un mecanismo de recuperación de errores para continuar con el análisis.



El análisis sintáctico se puede considerar como una función que toma como entrada la secuencia de componentes léxicos producida por el análisis léxico y produce como salida el árbol sintáctico. En la realidad, el análisis sintáctico hace una petición al análisis léxico del componente léxico siguiente en la entrada (los símbolos terminales) conforme lo va necesitando en el proceso de análisis, conforme se mueve a lo laro de la gramática.

Reconocedor versus analizador sintáctico

Un reconocedor trata de determinar simplemente si la cadena puede o no ser generada por la gramática (salida boolena). Podríamos preguntar algo más. Si queremos reconocer las estructuras propias de los lenguajes de programación en el fichero de entrada para guiar la traducción, es necesario conocer su estructura jerárquica, el árbol de análisis sintáctico.

3.2. Especificación sintáctica de los lenguajes de programación

La mayoría de las construcciones de los lenguajes de programación se pueden representar con una gramática independiente del contexto (GIC). La mayoría de las construcciones de los lenguajes de programación implican recursividad y anidamientos.

$$G = \{S, V_T, V_{NT}, P\},$$

S: el axioma o símbolo de inicio
V_T: conjunto de terminales, los componentes léxicos
V_{NT}: conjunto de no-terminales
P: conjunto de reglas de producción de la forma
$V_{NT} \rightarrow X_1, \ldots, X_n,\ con\ X_i \in (V_T \cup V_{NT})$
Pregunta: Si las expresiones regulares son un caso particular de GIC, **por qué no se incluye la especificación léxica como parte de la sintaxis**?

Los AFD son muy sencillos de implementar y son muy eficientes frente a los autómatas a pila necesarios para reconocer las GIC, la eficiencia del traductor se vería comprometida.

¿Expresiones regulares o gramáticas independientes del contexto?

Las expresiones regulares no permiten construcciones anidadas tan comunes en los lenguajes de programación: paréntesis equilibrados, concordancia de pares de palabras clave como `begin-end`, `do-while`, `...`

Por ejemplo: consideremos el problema de los paréntesis equilibrados en una expresión aritmética. El hecho de que haya un

paréntesis abierto obliga a que haya un paréntesis cerrado. Este problema es similar a considerar el lenguaje

$L = \{aba, aabaa, aaabaaa, \ldots, \} = \{a^n ba^n \leq 0\}$

Si intentamos escribir una expresión regular, lo más próximo sería: $\{a^*ba^*\}$ pero no se garantiza que el número de a's antes y después sea el mismo.

Las expresiones regulares NO SABEN contar. NO es posible especificar la estructura de un lenguaje de programación con sólo expresiones regulares.

¿Son suficientes las gramáticas independientes del contexto?

NO, existen ciertos aspectos de los lenguajes de programación que no se pueden representar con una GIC. A estos aspectos se les llama aspectos *semánticos*, son aspectos en general dependientes del contexto.

Por ejemplo, el problema de declaración de los identificadores antes de uso. Se podría representar mediante el lenguaje

L=$\{wcw \; tal \; que \; w \in (a|b)^*\}$

este lenguaje no puede ser representado por una GIC.

Por qué no usar gramáticas contextuales?

Su reconocimiento es demasiado complejo para ser eficiente en un problema práctico.

Será en el tema de análisis semántico donde se abordará estos problemas mediante el uso de acciones concretas (acciones semánticas) que no son propias de la gramática en sí. Será codigo escrito para hacer las comprobaciones semánticas necesarias (tipos, declaración antes de uso, ...)

Ejemplo de una gramática para un sencillo lenguaje:

Program →	**module id ;** Declarations **begin** StatementList **end id .**
Declarations →	**type id =** TypeDenoter **;** TypeList Declarations |
	var id IdList **:** TypeDenoter **;** VarList Declarations |
	function id (Parameter ParameterList **) :** SimpleType **;** Declarations
	begin StatementList **end id ;** Declarations | ϵ
TypeList →	**id =** TypeDenoter **;** TypeList | ϵ
VarList →	**id** IdList **:** TypeDenoter **;** VarList | ϵ
IdList →	**,** **id** IdList | ϵ
Parameter →	TypeDenoter **id** | ϵ
ParameterList →	**,** Parameter ParameterList | ϵ
SimpleType →	**char** | **integer** | **real** | **boolean**
TypeDenoter →	SimpleType | **id** | CompositeType
CompositeType →	**array** [**num_integer .. num_integer**] **of** SimpleType |
	record id IdList **:** SimpleType **;** FieldList **end**
FieldList →	**id** IdList **:** SimpleType **;** FieldList | ϵ
StatementList →	Statement **;** StatementList | ϵ
Statement →	AssignStm | IfStm | WhileStm | FunctionStm | ReturnStm | WriteStm | ReadStm
AssignStm →	LeftValue = Expression
LeftValue →	**id** | **id** [Expression] | **id . id**
IfStm →	**if** Expression **then** StatementList ElseStm **end**
ElseStm →	**else** StatementList | ϵ
WhileStm →	**while** Expression **do** StatementList **end**
FunctionStm →	**id (** Arg ArgList **)**
ReturnStm →	**return** Expression
WriteStm →	**write** Expression
ReadStm →	**read** LeftValue
Arg →	Num | **-** Num | **id** | **-** **id** | ϵ
Num →	**num_integer** | **num_real**
ArgList →	**,** Arg ArgList | ϵ
Expression →	LeftValue | (Expression) | **true** | **false** | Num | FunctionStm | Expression ***** Expression |
	Expression / Expression | Expression **-** Expression | Expression + Expression |
	- Expression | Expression **or** Expression | Expression **and** Expression | **not** Expression |
	Expression < Expression | Expression > Expression | Expression == Expression

Cuadro 3.1: Ejemplo de especificación sintáctica

3.3. Derivaciones. Árboles de anal. sintáctico

Dada $\alpha, \beta \in (V_T \cup V_{NT})^*$ cadenas arbitrarias de símbolos gramaticales, se dice que

$\alpha\ A\ \beta \overset{*}{\Rightarrow} \alpha\ \gamma\ \beta$ es una *derivación*, si existe una producción $A \rightarrow \gamma$.
Sea $S \overset{*}{\Rightarrow} \alpha$,
si $\alpha \in (V_T \cup V_{NT})^*$ se dice que α es una *forma de frase o forma sentencial*
si $\alpha \in (V_T)^*$ se dice que α es una *frase o sentencia.*
Existen dos tipos de derivaciones:

- *Derivación más a la izquierda*: se sustituye en cada paso el no-terminal más a la izquierda de la forma de frase. La denotaremos por:
  $\alpha \overset{*}{\underset{mi}{\Rightarrow}} \beta$

- *Derivación más a la derecha*: se sustituye en cada paso el no-terminal más a la derecha de la forma de frase. La denotaremos por:
  $\alpha \overset{*}{\underset{md}{\Rightarrow}} \beta$

Por ejemplo para la gramática:

E → **id** | **num** | E + E | (E) | - E

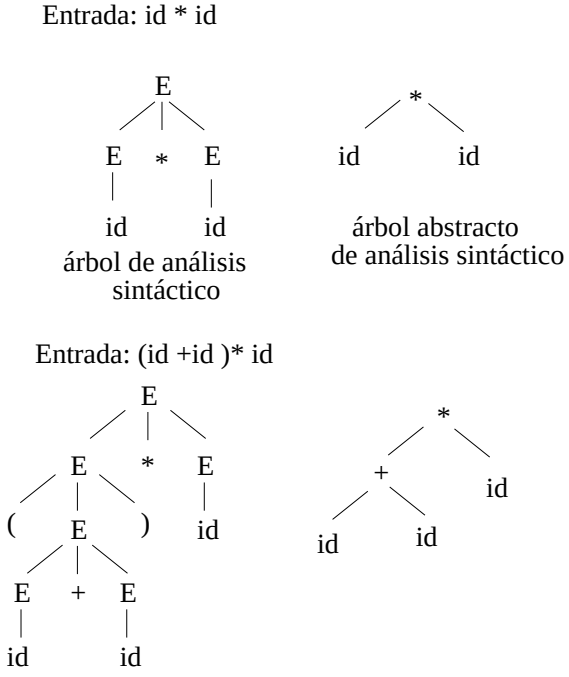
Si queremos derivar la frase $-(id + id)$
$E \Rightarrow -E \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id)$
derivación más a la izquierda.
$E \Rightarrow -E \Rightarrow -(E + E) \Rightarrow -(E + id) \Rightarrow -(id + id)$
derivación más a la derecha.

3.3.1. Árboles de análisis sintáctico

Un árbol de análisis sintáctico indica cómo a partir del axioma de la gramática se deriva una frase (cadena) del lenguaje. Dada una gramática independiente del contexto, un *árbol de análisis sintáctico* es un árbol tal que:

1. La raíz está etiquetada con el símbolo inicial.
2. Cada hoja está etiquetada con un componente léxico. Las hojas de izquierda a derecha forman la frase (el programa fuente).
3. Cada nodo interior está etiquetado con un no-terminal.
4. Si A es un no-terminal y $X_1, X_2, \ldots, X_n$ son sus hijos de izquierda a derecha, entonces existe la producción $A \rightarrow X_1, X_2, \ldots, X_n$, con $X_i \in (V_T \cup V_{NT})$.

El árbol de análisis sintáctico contiene en general mucha más información que la estrictamente necesaria para generar el código. Se puede construir una estructura más sencilla, los *árboles abstractos de análisis sintáctico*. Ejemplo: expresiones aritméticas (igual semántica, menor complejidad)



3.4. Gramáticas limpias y bien formadas

Llamamos *símbolo vivo* al símbolo a partir del cual se puede derivar una cadena de terminales. Llamamos *símbolo muerto* a los símbolos no-vivos, no generan una cadena del lenguaje. Llamamos *símbolo inaccesible* si nunca aparece en la parte derecha de una producción. A las gramáticas que contienen estos tipos de símbolos se les llama *gramáticas sucias*.

> *Teorema 1:* si todos los símbolos de la parte derecha de una producción son símbolos vivos, entonces el símbolo de la parte izquierda también lo es.

Algoritmo para detectar símbolos muertos

1. Hacer una lista de no-terminales que tengan al menos una producción con sólo símbolos terminales en la parte derecha.
2. Dada una producción, si todos los no-terminales de la parte derecha pertenecen a la lista, entonces podemos incluir en la lista al no-terminal de la parte izquierda de la producción.
3. Cuando ya no se puedan incluir más símbolos en la lista mediante la aplicación del paso 2, la lista contendrá los símbolos no-terminales vivos y el resto serán símbolos muertos.

Teorema 2: si el símbolo no-terminal de la parte izquierda de una producción es accesible, entonces todos los símbolos de la parte derecha también lo son.

Algoritmo para detectar símbolos inaccesibles

1. Se inicializa una lista de no-terminales que sabemos que son accesibles con el axioma.
2. Si la parte izquierda de una producción está en la lista, entonces se incluye en la misma al no-terminal que aparece en la parte derecha de la producción.
3. Cuando ya no se pueden incluir más símbolos a la lista mediante la aplicación del paso 2, entonces la lista contendrá todos los símbolos accesibles y el resto de los no-terminales serán innacesibles.

Para limpiar una gramática primero se eliminan los símbolos muertos y después los símbolos inaccesibles.
Una gramática *está bien formada* si es limpia y además no contiene producciones-ϵ. Para eliminar producciones-ϵ, remitirse a la signatura de TALF.
Ejemplo: limpiar la gramática

$$
\begin{aligned}
&S \rightarrow a\ A\ B \mid A\\
&A \rightarrow c\ B\ d\\
&B \rightarrow e \mid f\ S\\
&C \rightarrow g\ D \mid h\ D\ t\\
&D \rightarrow x \mid y \mid z
\end{aligned}
$$

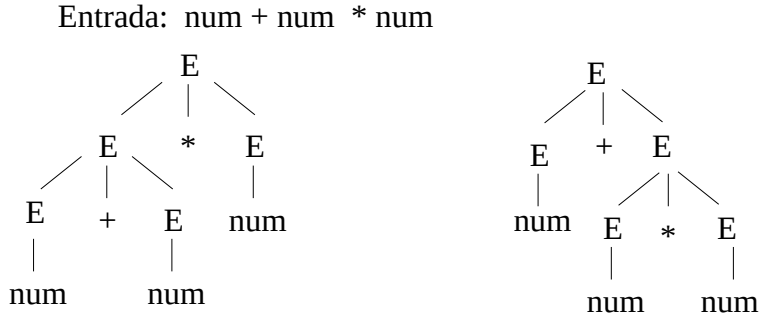
3.5. Gramáticas ambiguas

Una gramática es *ambigua* cuando para una determinada sentencia produce más de un árbol de derivación.

La gramática siguiente es ambigua:

E → **id** | **num** | E + E | E * E | (E) | - E

Supongamos la sentencia `id + id * id`



El significado semántico es DIFERENTE. No existe una única traducción posible. Se genera código diferente.
No existe un algoritmo que determine con certeza en un plazo finito de tiempo si una gramática es ambigua o no. A lo sumo que se ha llegado en el estudio de la ambigüedad es que hay algunas condiciones que de cumplirse determinan la ambigüedad, pero en el caso de no cumplirse no se puede decir que la gramática es no ambigua.
Necesidad de evitar las gramáticas ambiguas. **Cómo**?

- transformando la gramática o
- estableciendo precedencias entre operadores y de asociatividad.

Se puede eliminar la ambigüedad transformando la gramática agrupando todos los operadores de igual precedencia en grupos y asociando a cada uno una regla, de forma que los que tengan menor precedencia aparezcan más cercanos al símbolo de inicio, precedencia en cascada. Esto conlleva el aumento de la complejidad de la gramática y con ello en la del árbol sintáctico. La gramática deja de ser intuitiva.

Ejemplo 1: gramática ambigua

E → **num** | E + E | E - E | E * E | E / E

Si la transformamos, esta gramática ya no es ambigua.

exp → exp + term | exp - term | term
term → term * factor | term / factor | factor
factor → (exp) | **num**

IMPORTANTE: Una regla recursiva a izquierdas, hace que el operador sea asociativo por la izquierda. Una regla recursiva a derechas hace que el operador sea asociativo por la derecha.
Ejemplo 2: el problema del *dangling else* y el convenio de la asociación al `if` más cercano.

statement → if_stmt | **other**
if_stmt → **if** (exp) statement
| **if** (exp) statement **else** statement
exp → **0** | **1**

Por ejemplo: para la sentencia `if (0) if (1) other else other` tenemos dos árboles de derivación.

Crear los dos árboles ...

3.6. Clasificación métodos de análisis sintáctico

Atendiendo a la forma en que se construye el árbol de análisis sintáctico los métodos de análisis sintáctico se clasifican:

- *Métodos descendentes:* se parte del símbolo de inicio de la gramática que se coloca en la raíz y se va construyendo el árbol desde arriba hacia abajo hasta las hojas, eligiendo la derivación que da lugar a una concordancia con la cadena de entrada. Se basa en la idea de *predice* una derivación y establece una *concordancia* con el símbolo de la entrada (*predict/match*). El análisis sintáctico descendente corresponde con un recorrido prefijo del árbol de análisis sintáctico (primero expandimos el nodo que visitamos y luego procesamos los hijos).

  S → Sujeto Verbo Objeto
  Sujeto → el Nombre | un Nombre
  Objeto → el Nombre | un Nombre
  Verbo → el Nombre | un Nombre
  Nombre → perro | gato | león

  Ejemplo: `El león cazó un gato`

  Crear el árbol...

  Problemas: recursión a izquierdas, diferentes alternativas en una producción, ¿cómo conseguir un coste lineal?

- *Métodos ascendentes:* se construye el árbol de análisis sintáctico desde las hojas hasta la raíz. En las hojas está la cadena a analizar y se intenta reducirla al símbolo de inicio de la gramática que está en la raíz. Se trata de *desplazar*-se en la cadena de entrada y encontrar una subcadena para aplicar una *reducción* (una regla de producción a la inversa), (*shift-reduce*). El análisis sintáctico ascendente corresponde con un recorrido postorden del árbol (primero reconocemos los hijos y luego mediante una reducción reconocemos el padre).

Insertar un ejemplo de como efectivamente los dos tipos de análisis corresponden con los tipos de recorrido prefijo y postfijo.

Atendiendo a la forma en que procesan la cadena de entrada se clasifican en:

- *Métodos direccionales:* procesan la cadena de entrada símbolo a símbolo de izquierda a derecha.
- *Métodos no-direccionales:* acceden a cualquier lugar de la cadena de entrada para construir el árbol. Necesitan tener toda la cadena de componentes léxicos en memoria. Más costosos, no se suelen implementar.

Atendiendo al número de alternativas posibles en una derivación se clasifican en:

- *Métodos deterministas:* dado un símbolo de la cadena de entrada se puede decidir en cada paso cuál es la alternativa/derivación adecuada a aplicar, sólo hay una posible. No se produce retroceso y el coste el lineal.
- *Métodos no-deterministas:* en cada paso de la construcción del árbol se deben probar diferentes alternativas/derivaciones para ver cual es la adecuada, con el correspondiente aumento del coste.

Importancia de los métodos direccionales y deterministas en el diseño de traductores. El determinismo por la necesidad de eficiencia (coste lineal frente a exponencial que le haría prohibitivo en el diseño práctico de traductores) y porque las herramientas para la generación de traductores asumen esta condición. Los

	descendentes		ascendentes
no direccionales	Unger		CYK
direccionales	no deterministas	Predice/Concuerda 1º en anchura 1º en profundida	Desplaza/Reduce 1º en anchura 1º en profundida
	deterministas	Predice/Concuerda Gram. LL(1)	Desplaza/Reduce Gram. LR(k) LR(0), SLR(1), LALR(1)

métodos direccionales por la propia naturaleza secuencial en que se va procesando los símbolos del fichero fuente de izquierda a derecha.

3.7. Ejercicios

1. (0.2 ptos) Dadas la siguiente gramática, elimina los símbolos muertos e inaccesibles.

   S → a A B C | D d
   A → a B C
   B → b
   C → c
   D → d
   E → g h

2. (0.2 ptos) Dada la siguiente grámatica, escribe la derivación más a la izquierda, el árbol de análisis sintáctico y el árbol sintáctico abstracto para las siguientes expresiones: (a) 3+4*5-6; (b) 3*(4-5+6)

   exp → exp addop term | term
   addop → + | -
   term → term mulop factor | factor
   mulop → *| /
   factor → (exp) | **num**

3. (0.2 ptos) Dada la siguiente grámatica simplificada de una expresión en LISP: (a) escribe las derivaciones más a la derecha y más a la izquierda de la entrada $(a\ 23\ (m\ x\ y))$. Dibuja el árbol de análisis sintáctico para esa sentencia.

   lexp → atom | list
   atom → **number** | **identifier**
   list → (lexp-seq)
   lexp-seq → lexp-seq lexp | lexp

4. (0.2 ptos) Dada la siguiente gramática, construya el árbol de análisis sintáctico para la frase `not (true or false)`. ¿Es una gramática ambigua?

bexp → bexp **or** bterm | bterm
bterm → bterm **and** bfactor | bfactor
bfactor → **not** bfactor | (bexp) | **true** | **false**

5. (0.2 ptos) Escribe una gramática para expresiones booleanas que incluya las constantes `true` y `false`, los operadores `and`, `or` y `not` y los paréntesis. Se ha de tener en cuenta que el operador `or` tiene menor precedencia que el operador `and` y éste menor que el operador `not`. Construye la gramática de forma que los operadores `and` y `or` sean asociativos por la izquierda y el operador `not` por la derecha.

6. (0.2 ptos) Comprueba que se puede resolver el problema de la ambigüedad del *dangling else* transformando la gramática a la forma:

statment → matched-stmt | unmatched-stmt
matched-stmt → **if** (exp) matched-stmt **else** matched-stmt | **other**
unmatched-stmt → **if** (exp) statement
| **if** (exp) matched-stmt **else** unmatched-stmt
exp → 0 | 1

Justificar usando un ejemplo.

7. (0.2 ptos) Comprueba que el intento de resolver el problema de la ambigüedad del *dangling else* no es adecuado:

statment → if (exp) statement | matched-stmt
matched-stmt → if (exp) matched-stmt else statment | other
exp → 0 | 1

Justificar usando un ejemplo.