Universidad de Huelva

Tema 3

Análisis sintáctico descendente

Índice

3.1 Características del análisis sintáctico

3.2 Gramáticas libres de contexto

3.3 Notación EBNF

3.4 Tipos de análisis sintáctico

3.5 Análisis sintáctico descendente predictivo

3.6 La condición LL(1)

3.7 ASDP dirigido por tabla

3.8 Analizador Descendente Recursivo

3.9 Gestión de errores

Índice

3.1 Características del análisis sintáctico

3.2 Gramáticas libres de contexto

3.3 Notación EBNF

3.4 Tipos de análisis sintáctico

3.5 Análisis sintáctico descendente predictivo

3.6 La condición LL(1)

3.7 ASDP dirigido por tabla

3.8 Analizador Descendente Recursivo

3.9 Gestión de errores

- Lee componentes léxicos (tokens)
- Comprueba que el orden de estos corresponde a la sintaxis predeterminada
- Genera errores en caso de que el flujo de tokens no responda a la sintaxis
- Genera árboles de análisis sintáctico
- Se suele conocer como “Parser”

- El análisis sintáctico desarrolla el esqueleto de toda la fase de análisis
- Utiliza el analizador léxico como una rutina dentro del análisis sintáctico (getNextToken())
- Integra el análisis semántico como un conjunto de rutinas a ejecutar durante la comprobación de la sintaxis

- Aunque el objetivo teórico es construir un árbol de análisis sintáctico, este raramente se construye como tal, sino que las rutinas semánticas integradas van generando el árbol de sintaxis abstracta.
- El análisis sintáctico se especifica mediante una gramática libre de contexto
- El análisis sintáctico se implementa mediante un autómata de pila

Índice

3.1 Características del análisis sintáctico

3.2 Gramáticas libres de contexto

3.3 Notación EBNF

3.4 Tipos de análisis sintáctico

3.5 Análisis sintáctico descendente predictivo

3.6 La condición LL(1)

3.7 ASDP dirigido por tabla

3.8 Analizador Descendente Recursivo

3.9 Gestión de errores

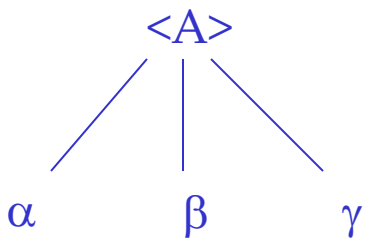
- Definición: (N, Σ, P, <S>)
  - N: Alfabeto de símbolos no terminales
  - Σ: Alfabeto de símbolos terminales
  - P: Conjunto de producciones o reglas

$$P \subset N \times (N \cup \Sigma)^*$$

  - <S>: Símbolo inicial (no terminal)

- Las gramáticas regulares no son adecuadas para describir la sintaxis de lenguajes de programación
  - [Ejemplo] sintaxis de llaves abiertas y cerradas: {{{{ a }}}}
  - El número de llaves abiertas y cerradas debe ser el mismo
  - Es indispensable para analizar la sintaxis de C, C++ y Java (p. e.)
  - [Expresión regular] : ('{')* a ('}')*
  - Puede generar un número de llaves diferente
- Las gramáticas libres de contexto sí permiten este tipo de sintaxis
  - <A> → '{' <A> '}'
  - <A> → a

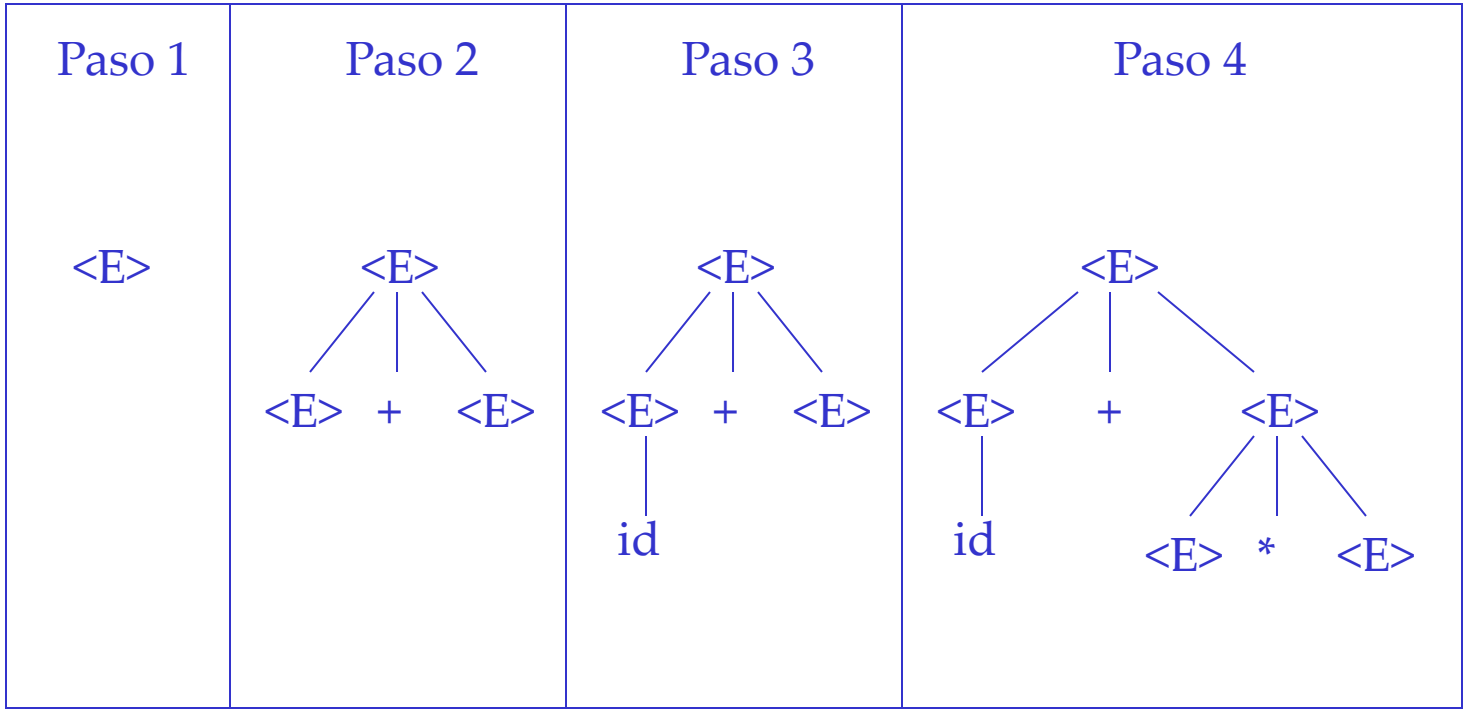
- Árboles de análisis sintáctico
  - Los nodos corresponden a símbolos no terminales
  - Las hojas corresponden a símbolos terminales y no terminales
  - La ramificación describe una producción
- Ejemplo de producción: <A> → α β γ
- Ejemplo de árbol de análisis sintáctico



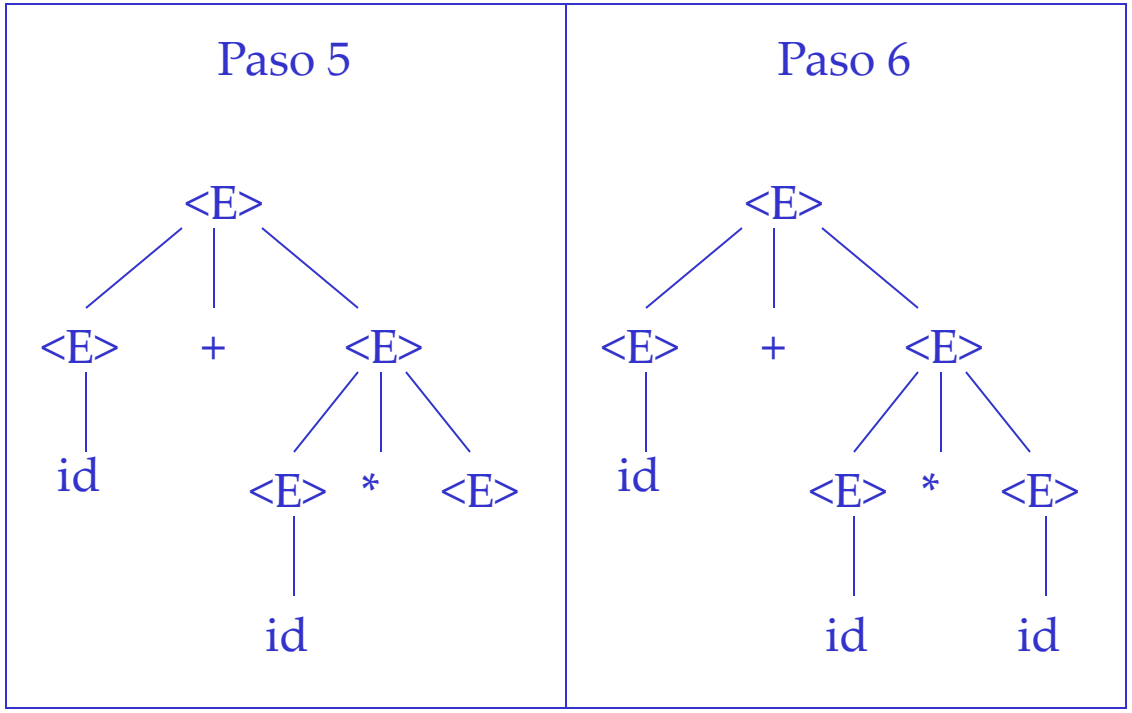
- Derivación de árboles de análisis sintáctico
  - Sucesión de árboles de análisis sintáctico
  - El primer árbol representa el símbolo inicial
  - Cada árbol se diferencia del anterior en la expansión de una hoja no terminal por medio de una producción
  - Las hojas del árbol final son todas símbolos terminales
  - Las formas sentenciales se obtienen leyendo las hojas de izquierda a derecha

- Ejemplo de derivación de árboles de análisis sintáctico
- Gramática:
  - <E> $\rightarrow$ <E> + <E>
  - <E> $\rightarrow$ <E> * <E>
  - <E> $\rightarrow$ (<E>)
  - <E> $\rightarrow$ id
- Sentencia:
  - id + id * id

- Ejemplo de derivación de árboles de análisis sintáctico

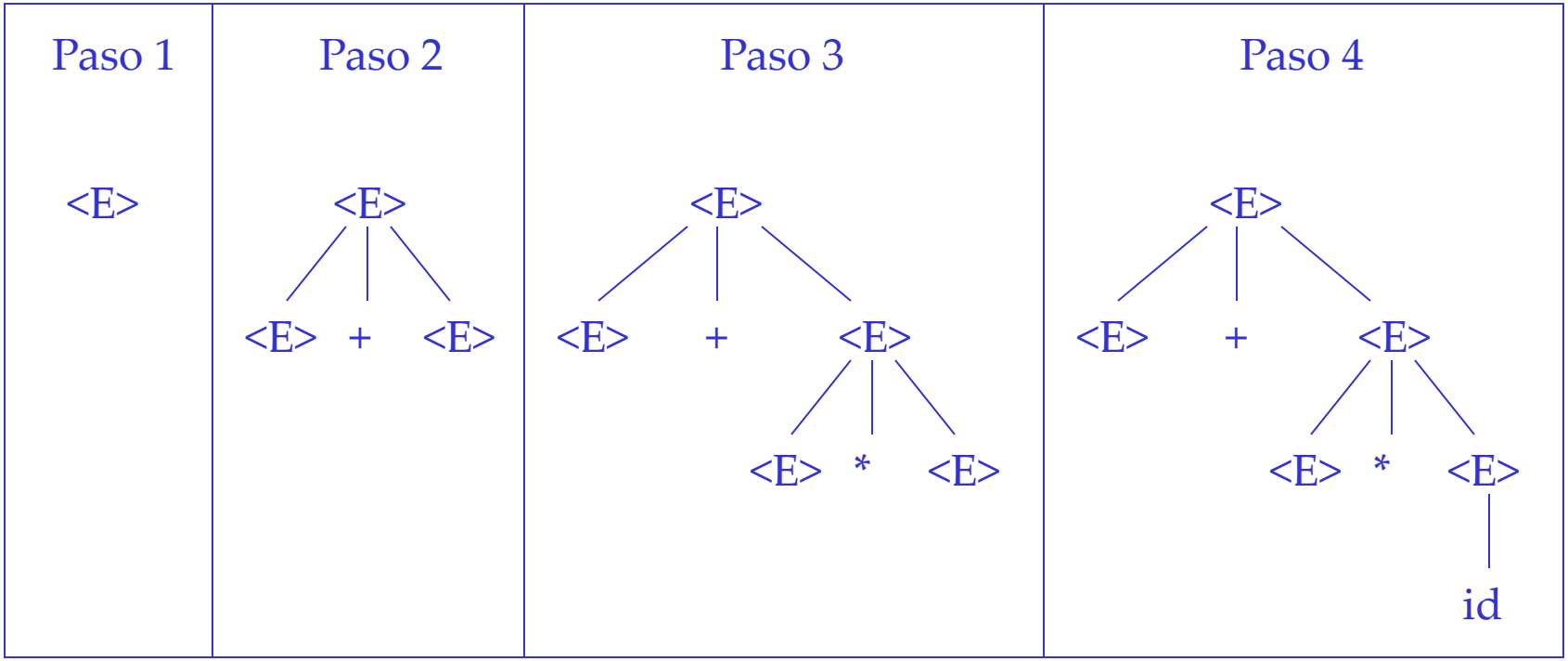


- Ejemplo de derivación de árboles de análisis sintáctico

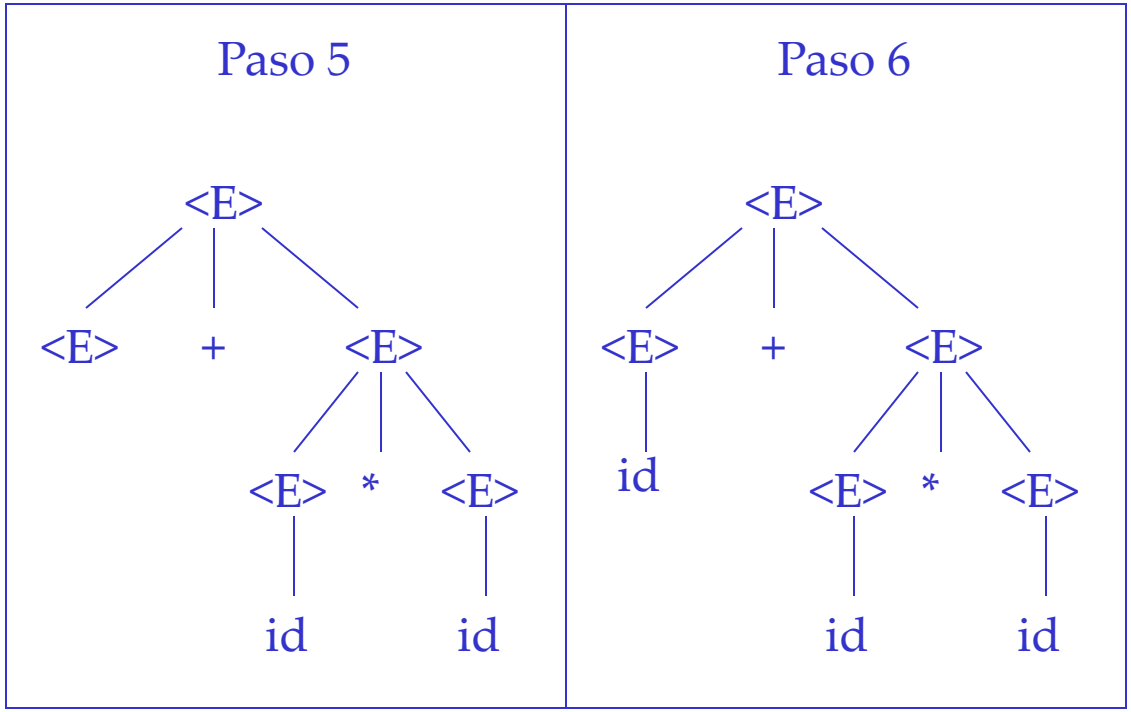


- Derivación canónica:
  - Aquella que sigue un orden predeterminado para sustituir las hojas no terminales
- Derivación canónica a la izquierda:
  - Se sustituye siempre la hoja no terminal que se encuentre más a la izquierda
  - El ejemplo anterior es una derivación a la izquierda
- Derivación canónica a la derecha:
  - Se sustituye siempre la hoja no terminal que se encuentre más a la derecha

- Ejemplo de derivación canónica a la derecha



- Ejemplo de derivación canónica a la derecha

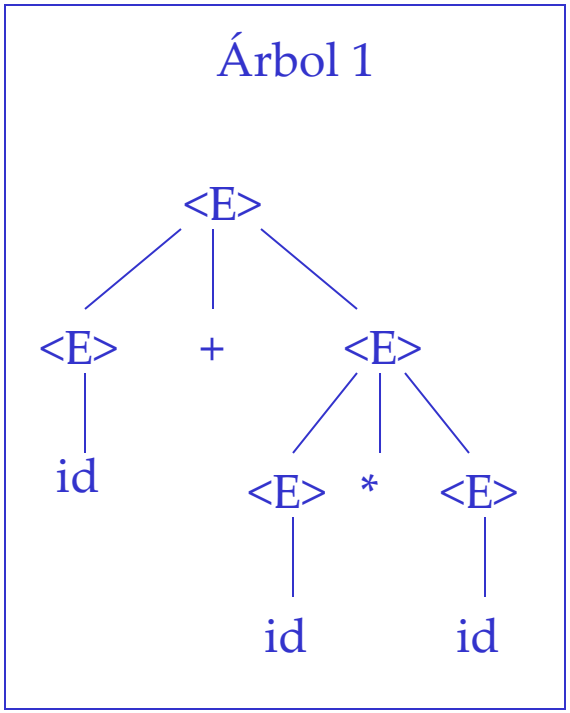


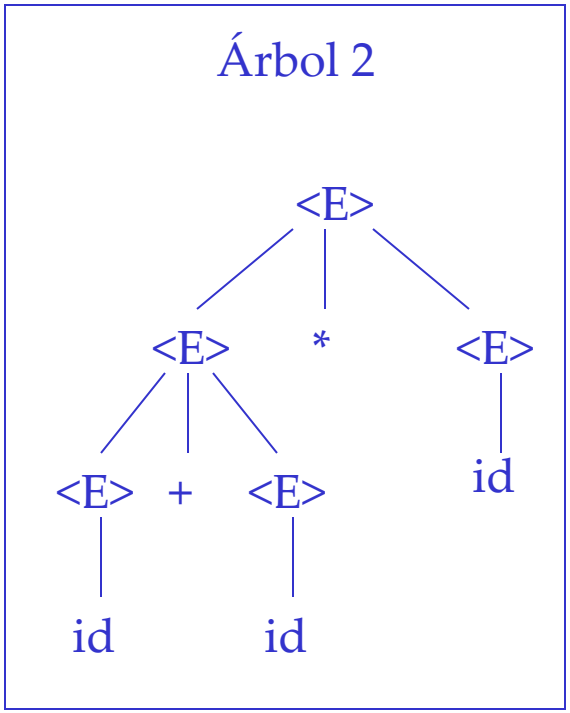
- Ambigüedad:
  - Una gramática es ambigua si existe alguna cadena a la que se pueda asociar dos árboles de análisis sintáctico diferentes
  - Las gramáticas ambiguas no son adecuadas para especificar lenguajes de programación, ya que árboles sintácticos diferentes generan un código diferente.
  - Demostrar que una gramática no es ambigua es muy difícil
  - Existen construcciones gramaticales que producen ambigüedad. Si se usan estas construcciones es fácil demostrar que una gramática es ambigua.

- Construcciones que provocan ambigüedad:
  - Reglas de la forma
    - <E> $\rightarrow$ <E> ..... <E>
  - Gramáticas con ciclos
    - <S> $\rightarrow$ <A>
    - <S> $\rightarrow$ a
    - <A> $\rightarrow$ <S>
  - Conjuntos de reglas con caminos alternativos entre dos puntos
    - <S> $\rightarrow$ <A>
    - <S> $\rightarrow$ <B>
    - <A> $\rightarrow$ <B>

- Construcciones que provocan ambigüedad:
  - Producciones recursivas en las que la regla no recursiva pueda producir la cadena vacía
    - <S> → <H><R><S>
    - <S> → s
    - <H> → h | λ
    - <R> → r | λ
  - Símbolos no terminales que puedan producir la misma cadena y la cadena vacía, si aparecen juntos
    - <S> → <H><R>
    - <H> → h | λ
    - <R> → h | r | λ

- Ejemplo de ambigüedad
  - Sentencia: id + id * id





- Recursividad:
  - Una gramática es recursiva si a partir de un símbolo no terminal se puede obtener una forma sentencial que incluya el mismo símbolo no terminal
    - <A> ⇨ α <A> β
  - Una gramática es recursiva a la izquierda si en alguna recursividad, el símbolo no terminal aparece a la izquierda de la forma sentencial
    - <A> ⇨ <A> β
  - Una gramática es recursiva a la derecha si en alguna recursividad, el símbolo no terminal aparece a la derecha de la forma sentencial
    - <A> ⇨ α <A>

Índice

3.1 Características del análisis sintáctico

3.2 Gramáticas libres de contexto

3.3 Notación EBNF

3.4 Tipos de análisis sintáctico

3.5 Análisis sintáctico descendente predictivo

3.6 La condición LL(1)

3.7 ASDP dirigido por tabla

3.8 Analizador Descendente Recursivo

3.9 Gestión de errores

- Significa "*Forma de Backus-Naur Extendida*"
- Permite definir *Gramáticas con partes derechas regulares*
- Extensiones:
  - Disyunción: $(\alpha_1 \mid \alpha_2 \mid \alpha_3)$
  - Clausura: $(\alpha)^*$
  - Clausura positiva: $(\alpha)^+$
  - Opcionalidad: $(\alpha)?$
- Estas extensiones no añaden poder expresivo, es decir, las gramáticas con partes derechas regulares tienen la misma capacidad expresiva que las gramáticas libres de contexto

- Transformación de la notación EBNF en BNF
  - Disyunción: $\langle A \rangle \rightarrow \alpha\ (\rho_1 \mid \rho_2 \mid \rho_3)\ \beta$
    - $\langle A \rangle \rightarrow \alpha\ \langle A' \rangle\ \beta$
    - $\langle A' \rangle \rightarrow \rho_1$
    - $\langle A' \rangle \rightarrow \rho_2$
    - $\langle A' \rangle \rightarrow \rho_3$
  - Clausura: $\langle A \rangle \rightarrow \alpha\ (\ \rho\ )^*\ \beta$
    - $\langle A \rangle \rightarrow \alpha\ \langle A' \rangle\ \beta$
    - $\langle A' \rangle \rightarrow \rho\ \langle A' \rangle$
    - $\langle A' \rangle \rightarrow \lambda$

- Transformación de la notación EBNF en BNF
  - Clausura positiva: $\langle A \rangle \rightarrow \alpha (\rho)+ \beta$
    - $\langle A \rangle \rightarrow \alpha \rho \langle A' \rangle \beta$
    - $\langle A' \rangle \rightarrow \rho \langle A' \rangle$
    - $\langle A' \rangle \rightarrow \lambda$
  - Opcionalidad: $\langle A \rangle \rightarrow \alpha (\rho)? \beta$
    - $\langle A \rangle \rightarrow \alpha \langle A' \rangle \beta$
    - $\langle A' \rangle \rightarrow \rho$
    - $\langle A' \rangle \rightarrow \lambda$

- Reescritura de expresiones en notación EBNF

Notación BNF

- <E> → <T> <E'>
- <E'> → + <T> <E'>
- <E'> → λ
- <T> → <F> <T'>
- <T'> → * <F> <T'>
- <T'> → λ
- <F> → num
- <F> → (<E>)

Notación EBNF

- <E> → <T> ('+' <T>)*
- <T> → <F> ('*' <F>) *
- <F> → num | '(' <E> ')'

Índice

3.1 Características del análisis sintáctico

3.2 Gramáticas libres de contexto

3.3 Notación EBNF

3.4 Tipos de análisis sintáctico

3.5 Análisis sintáctico descendente predictivo

3.6 La condición LL(1)

3.7 ASDP dirigido por tabla

3.8 Analizador Descendente Recursivo

3.9 Gestión de errores

- Problema del análisis:
  - Dada una cadena x
  - Dada la gramática G
  - ¿ Pertenece x al lenguaje reconocido por G ? ¿ $x \in L(G)$?
- Algoritmos de resolución general
  - Algoritmo de Cocke-Younger-Kasami: Orden $|x|^3$
  - Algoritmo de Early: $O(|x|^3)$ en gramáticas ambiguas y $O(|x|^2)$ en gramáticas no ambiguas
  - Para una compilación en un tiempo razonable es necesario un orden lineal
- Algoritmos de orden lineal
  - Imponen restricciones a las gramáticas a analizar
  - Análisis descendente
  - Análisis ascendente

- Análisis descendente
  - Se construye el árbol de análisis sintáctico partiendo de la raíz hacia las hojas
  - Se estudia los siguientes tokens a analizar para decidir la regla a expandir
  - Lookahead: Nº de tokens necesario para realizar la elección de la regla a expandir
  - Gramáticas LL(1): con Lookahead = 1

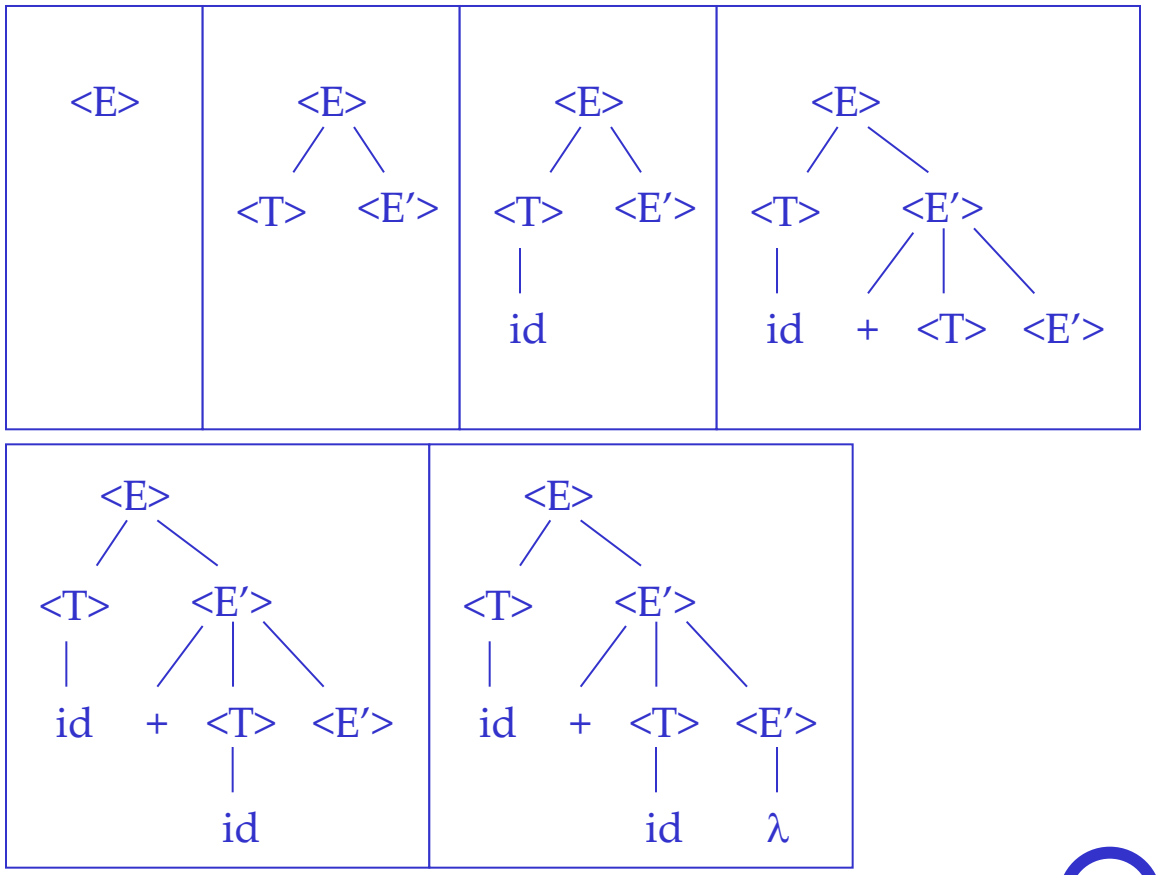
- Análisis descendente (ejemplo)

Gramática

- <E> → <T> <E′>
- <E′> → + <T> <E′>
- <E′> → λ
- <T> → id

Sentencia

id + id



- Análisis ascendente
  - Se construye el árbol de análisis sintáctico partiendo de las hojas hacia la raíz
  - Cuando se obtiene la parte derecha de una regla, se sustituye por su símbolo no terminal
  - Si se analiza la derivación del árbol de análisis sintáctico en orden inverso se obtiene una derivación a la derecha
  - Gramáticas LR

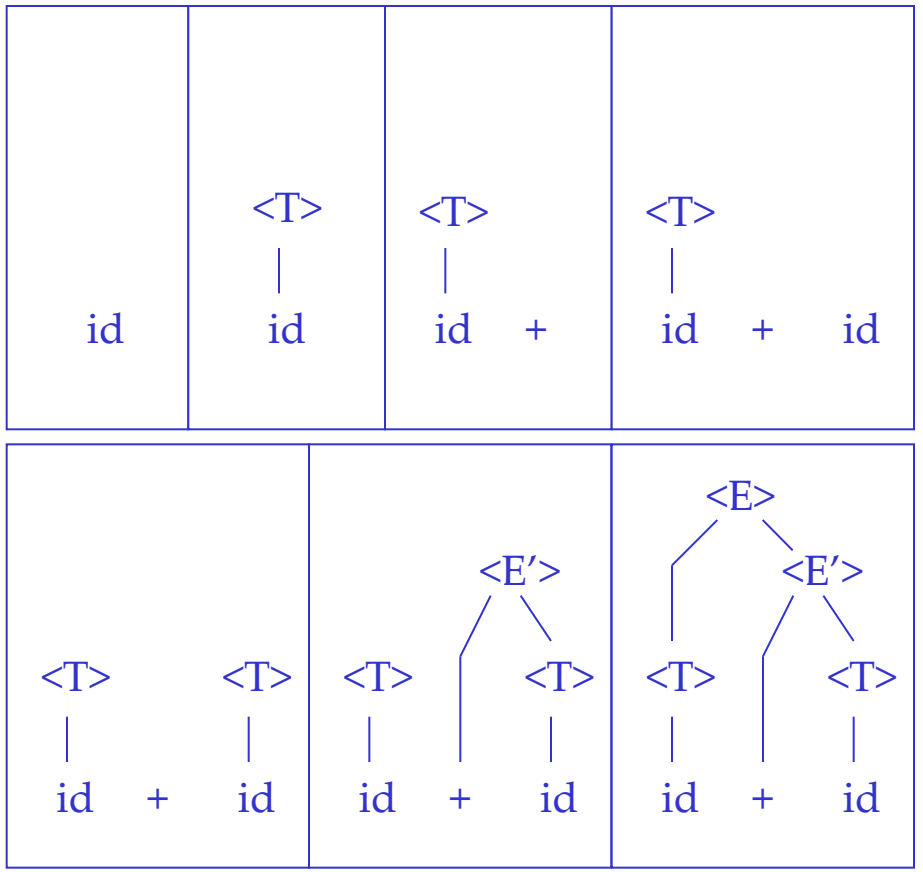
- Análisis ascendente (ejemplo)

Gramática

- <E> → <T> <E'>
- <E'> → + <T> <E'>
- <E'> → λ
- <T> → id

Sentencia

id + id



Índice

3.1 Características del análisis sintáctico

3.2 Gramáticas libres de contexto

3.3 Notación EBNF

3.4 Tipos de análisis sintáctico

3.5 Análisis sintáctico descendente predictivo

3.6 La condición LL(1)

3.7 ASDP dirigido por tabla

3.8 Analizador Descendente Recursivo

3.9 Gestión de errores

- Análisis sintáctico descendente con retroceso:
  - Consiste en intentar sustituir cada símbolo no terminal por cada una de sus producciones hasta encontrar la producción adecuada.
- Procedimiento:
  - Se parte del símbolo inicial
  - Se realiza una derivación por la izquierda
  - En cada paso se sustituye un no terminal por una de sus reglas de producción.
  - Si se produce un error se elimina la última derivación y se sustituye el no terminal por la siguiente regla de producción

- Ejemplo:
  - Gramática
    - <S> → c <A> d
    - <A> → ab
    - <A> → a
  - Cadena a analizar: **cad**
  - Procedimiento:
    - <S> ⇨ c <A> d [consumir c]
    - <S> ⇨ c <A> d ⇨ c a b d [consumir a] [consumir b] ERROR
    - <S> ⇨ c <A> d ⇨ c a d [consumir a] [consumir d] CORRECTO

- Características del análisis descendente con retroceso
  - Tiene una complejidad exponencial en el peor caso
  - Sólo resulta factible con gramáticas sencillas
  - Se suele utilizar con herramientas que desarrollan el retroceso de manera automática (por ejemplo, en Prolog)
  - Se utiliza en tratamiento de ficheros de configuración simples

- Análisis sintáctico descendente predictivo
  - El análisis con retroceso resulta de orden exponencial porque intenta sustituir cada no terminal con todas sus producciones.
  - Para obtener un análisis de orden lineal es necesario saber cual es la producción que debe utilizarse para expandir un símbolo no terminal.
  - Para ello es necesario conocer cuál es el token que se pretende analizar y elegir la producción a expandir
  - Dado un token y un símbolo no terminal, sólo debe existir una producción que comience por ese token (condición LL(1)).
  - La elección se basa en el uso de *conjuntos de predicción*

- El conjunto *primeros*(α)
  - Se aplica a formas sentenciales (α)
  - Definición formal:
    - **a** ∈ Prim(α) si **a** ∈ Σ y α ⇨ aβ
    - λ ∈ Prim(α) si α ⇨ λ

- Cálculo del conjunto *primeros*(α)
  - Definición: Conjunto Primeros de un símbolo no terminal
    - Es la unión del conjuto primeros de la parte derecha de sus reglas
    - $Prim(<A>) = \cup Prim(\alpha_i),\ <A> \rightarrow \alpha_i$
  - Si $\alpha \equiv \lambda$, $Prim(\alpha) = \{ \lambda \}$
  - Si $\alpha \equiv a_1\ a_2 \ldots a_n$
    - $a_1 \in \Sigma$, $Prim(\alpha) = \{ a_1 \}$
    - $a_1 \in N$, $\lambda \notin Prim(a_1)$, $Prim(\alpha) = Prim(a_1)$
    - $a_1 \in N$, $\lambda \in Prim(a_1)$, $Prim(\alpha) = \{ Prim(a_1) - \{ \lambda \} \} \cup Prim(a_2 \ldots a_n)$

- El conjunto *siguientes*(<A>)
  - Se aplica a símbolos no terminales, <A> ∈ N
  - Se refiere al conjunto de símbolos terminales (incluido el fin de entrada, $) que pueden aparecer a continuación de <A> en alguna forma sentencial.
  - Definición formal:
    - **a** ∈ Sig(<A>) si **a** ∈ Σ y <S> ⇨ α <A> **a** β
    - $ ∈ Sig(<A>) si <S> ⇨ α <A>

- Cálculo del conjunto *siguientes*(<A>)
  - Inicialmente
    - Sig(<A>) = ϕ
  - Si <A> es el símbolo inicial,
    - Sig(<A>) = Sig(<A>) $\cup$ { $ }
  - Para cada regla <B> $\rightarrow \alpha$ <A> β
    - Si $\lambda \notin$ Prim(β), Sig(<A>) = Sig(<A>) $\cup$ Prim(β)
    - Si $\lambda \in$ Prim(β), Sig(<A>) = Sig(<A>) $\cup$ {Prim(β) – { λ }} $\cup$ Sig(<B>)
  - Para cada regla <B> $\rightarrow \alpha$ <A>
    - Sig(<A>) = Sig(<A>) $\cup$ Sig(<B>)

- El conjunto *predicción*(<A> $\rightarrow \alpha$)
  - Se aplica a producciones, <A> $\rightarrow \alpha$
  - Se refiere al conjunto de símbolos terminales (incluido el fin de entrada, $, y excluido λ) que pueden aparecer cuando se va a expandir la producción <A> $\rightarrow \alpha$
  - Cálculo del conjunto:
    - Si $\lambda \notin$ Prim(α), Pred(<A> $\rightarrow \alpha$) = Prim(α)
    - Si $\lambda \in$ Prim(α), Pred(<A> $\rightarrow \alpha$) = { Prim(α) – { λ } } $\cup$ Sig(<A>)

- Ejemplo del cálculo de los conjuntos de predicción

Gramática

- <E> → <T> <E'>
- <E'> → + <T> <E'>
- <E'> → λ
- <T> → <F> <T'>
- <T'> → * <F> <T'>
- <T'> → λ
- <F> → num
- <F> → (<E>)

Conjunto Primeros

- Prim(<E>) = { num, (}
- Prim(<E'>) = { +, λ }
- Prim(<T>) = { num, (}
- Prim(<T'>) = { *, λ }
- Prim(<F>) = { num, (}

- Ejemplo del cálculo de los conjuntos de predicción

Gramática

- <E> → <T> <E'>
- <E'> → + <T> <E'>
- <E'> → λ
- <T> → <F> <T'>
- <T'> → * <F> <T'>
- <T'> → λ
- <F> → num
- <F> → (<E>)

Conjunto Siguientes

- Sig(<E>) = {), $ }
- Sig(<E'>) = Sig(<E>) = {), $ }
- Sig(<T>) =
  = { Prim(<E'>) – {λ} } ∪ Sig(<E'>)
  = { +,), $ }
- Sig(<T'>) = Sig(<T>) = { +,), $ }
- Sig(<F>) =
  = { Prim(<T'>) – {λ} } ∪ Sig(<T'>)
  = { *, + ,), $ }

- Ejemplo del cálculo de los conjuntos de predicción

Gramática

- <E> → <T> <E'>
- <E'> → + <T> <E'>
- <E'> → λ
- <T> → <F> <T'>
- <T'> → * <F> <T'>
- <T'> → λ
- <F> → num
- <F> → (<E>)

Conjunto de predicción

- Pred(<E> → <T> <E'>) = Prim(<T>) = { num, (}
- Pred(<E'> → + <T> <E'>) = Prim(+) = { + }
- Pred(<E'> → λ) = Sig(<E'>) = {), $ }
- Pred(<T> → <F> <T'>) = Prim(<F>) = { num, (}
- Pred(<T'> → * <F> <T'>) = Prim(*) = { * }
- Pred(<T'> → λ) = Sig(<T'>) = { +,), $ }
- Pred(<F> → num) = Prim(num) = { num }
- Pred(<F> → (<E>)) = Prim(( ) = { (}

3.1 Características del análisis sintáctico

3.2 Gramáticas libres de contexto

3.3 Notación EBNF

3.4 Tipos de análisis sintáctico

3.5 Análisis sintáctico descendente predictivo

3.6 La condición LL(1)

3.7 ASDP dirigido por tabla

3.8 Analizador Descendente Recursivo

3.9 Gestión de errores

- Es la condición que debe cumplirse para poder realizar un análisis descendente predictivo
- Para cada símbolo no terminal <A>
  - Dadas todas las producciones <A> $\rightarrow \alpha_i$
  - Los conjuntos Pred(<A> $\rightarrow \alpha_i$) deben ser disjuntos

Modificación de gramáticas no LL(1)

- No siempre es posible convertir una gramática no LL(1) en una gramática equivalente LL(1)
- Eliminación de la ambigüedad
  - Es la modificación más difícil y obliga a detectar ambigüedades y replantear el diseño de la gramática

Modificación de gramáticas no LL(1)

- Factorización por la izquierda
  - Transformar
    - $<A> \rightarrow \alpha\ \beta_1$
    - $<A> \rightarrow \alpha\ \beta_2$
  - En
    - $<A> \rightarrow \alpha\ <A'>$
    - $<A'> \rightarrow \beta_1$
    - $<A'> \rightarrow \beta_2$

Modificación de gramáticas no LL(1)

- Eliminación de la recursividad por la izquierda
  - Transformar
    - $<A> \rightarrow <A> \alpha$
    - $<A> \rightarrow \beta$
    - (genera el lenguaje: β, $\beta\alpha$, $\beta\alpha\alpha$, ...)
  - En
    - $<A> \rightarrow \beta <A'>$
    - $<A'> \rightarrow \alpha <A'>$
    - $<A'> \rightarrow \lambda$

Modificación de gramáticas no LL(1)

- Ejemplo
  - <Sent> → **if** <Expr> **then** <Sent> **else** <Sent> **endif**
  - <Sent> → **if** <Expr> **then** <Sent> **endif**
  - <Sent> → <Otras>
- Se transforma en
  - <Sent> → **if** <Expr> **then** <Sent> <Sent'>
  - <Sent> → <Otras>
  - <Sent'> → **else** <Sent> **endif**
  - <Sent'> → **endif**

Modificación de gramáticas no LL(1)

- Ejemplo
  - <E> → <E> + <T>
  - <E> → <E> - <T>
  - <E> → <T>
  - <T> → <T> * <F>
  - <T> → <T> / <F>
  - <T> → <F>
  - <F> → **num**
  - <F> → **(** <E> **)**

- Se transforma en
  - <E> → <T> <E′>
  - <E′> → + <T> <E′>
  - <E′> → - <T> <E′>
  - <E′> → λ
  - <T> → <F> <T′>
  - <T′> → * <F> <T′>
  - <T′> → / <F> <T′>
  - <T′> → λ
  - <F> → num
  - <F> → (<E>)

3.1 Características del análisis sintáctico

3.2 Gramáticas libres de contexto

3.3 Notación EBNF

3.4 Tipos de análisis sintáctico

3.5 Análisis sintáctico descendente predictivo

3.6 La condición LL(1)

3.7 ASDP dirigido por tabla

3.8 Analizador Descendente Recursivo

3.9 Gestión de errores

- Su funcionamiento se basa en una pila y una tabla de análisis
- La pila almacena las hojas del árbol de análisis sintáctico
- Siempre se estudia el contenido de la cima de la pila
- La tabla de análisis tiene
  - Una fila por cada símbolo no terminal
  - Una columna por cada símbolo terminal y el fin de entrada
- Las celdas de la tabla contienen enlaces a las reglas de producción
- El contenido de la tabla indica que regla expandir en función del símbolo terminal recibido

- Funcionamiento
  - Almacenar en la pila el símbolo de fin de entrada
  - Almacenar en la pila el símbolo inicial
  - Leer el siguiente token
  - Repetir hasta que la pila y la cadena de entrada estén vacías:
    - Si el símbolo es de la cima de la pila es un terminal entonces
      - Si el símbolo es el siguiente token, consumirlo y leer el siguiente token
      - Si el símbolo no es el siguiente token, generar error
    - Si el símbolo es un no terminal <A> entonces
      - Buscar en la tabla la fila <A> y la columna "siguiente token"
      - Si la celda contiene la regla (<A> → a1 a2 a3), almacenar a3, a2, a1.
      - Si la celda no contiene ninguna regla, generar error

- Contenido de la tabla de análisis
  - Para cada símbolo no terminal <A> se estudian sus producciones
  - Para cada producción (<A> $\rightarrow \alpha_i$) se estudia su conjunto de predicción, Pred(<A> $\rightarrow \alpha_i$)
  - Para cada elemento **a** del conjunto de predicción Pred(<A> $\rightarrow \alpha_i$) se incluye en la celda (<A>, a) el enlace a la producción (<A> $\rightarrow \alpha_i$)
  - Las celdas vacías suponen errores sintácticos.
  - Si la gramática no fuera LL(1) se producirían colisiones al asignar el valor a las celdas

- Ejemplo

Gramática

- <E> → <T> <E'>
- <E'> → + <T> <E'>
- <E'> → λ
- <T> → <F> <T'>
- <T'> → * <F> <T'>
- <T'> → λ
- <F> → num
- <F> → (<E>)

Conjunto de predicción

- Pred(<E> → <T> <E'>) = { num, (}
- Pred(<E'> → + <T> <E'>) = { + }
- Pred(<E'> → λ) = {), $ }
- Pred(<T> → <F> <T'>) = { num, (}
- Pred(<T'> → * <F> <T'>) = { * }
- Pred(<T'> → λ) = { +,), $ }
- Pred(<F> → num) = { num }
- Pred(<F> → (<E>)) = { (}

- Ejemplo

	num	+	*	(	)	$
<E>	→ <T> <E'>			→ <T> <E'>		
<E'>		→ + <T> <E'>			→ λ	→ λ
<T>	→ <F> <T'>			→ <F> <T'>		
<T'>		→ λ	→ * <F> <T'>		→ λ	→ λ
<F>	→ num			→ (<E>)		

- Ejemplo

Pila	Entrada	Acción
<E> $	num * (num + num) $	→ <T> <E'>
<T> <E'> $	num * (num + num) $	→ <F> <T'>
<F> <T'> <E'> $	num * (num + num) $	→ num
num <T'> <E'> $	num * (num + num) $	match(num)
<T'> <E'> $	* (num + num) $	→ * <F> <T'>
* <F> <T'> <E'> $	* (num + num) $	match(*)
<F> <T'> <E'> $	(num + num) $	→ (<E>)
(<E>) <T'> <E'> $	(num + num) $	match('(')
<E>) <T'> <E'> $	num + num) $	→ <T> <E'>
<T> <E'>) <T'> <E'> $	num + num) $	→ <F> <T'>
<F> <T'> <E'>) <T'> <E'> $	num + num) $	→ num

- Ejemplo

Pila	Entrada	Acción
num <T'> <E'>) <T'> <E'> $	num + num) $	match(num)
<T'> <E'>) <T'> <E'> $	+ num) $	→ λ
<E'>) <T'> <E'> $	+ num) $	→ + <T> <E'>
+ <T> <E'>) <T'> <E'> $	+ num) $	match('+')
<T> <E'>) <T'> <E'> $	num) $	→ <F> <T'>
<F> <T'> <E'>) <T'> <E'> $	num) $	→ num
num <T'> <E'>) <T'> <E'> $	num) $	match(num)
<T'> <E'>) <T'> <E'> $	) $	→ λ
<E'>) <T'> <E'> $	) $	→ λ
) <T'> <E'> $	) $	match(')')
<T'> <E'> $	$	→ λ

- Ejemplo

Pila	Entrada	Acción
<E'> $	$	→ λ
$	$	**match($)**
		aceptar

3.1 Características del análisis sintáctico

3.2 Gramáticas libres de contexto

3.3 Notación EBNF

3.4 Tipos de análisis sintáctico

3.5 Análisis sintáctico descendente predictivo

3.6 La condición LL(1)

3.7 ASDP dirigido por tabla

3.8 Analizador Descendente Recursivo

3.9 Gestión de errores

- Su funcionamiento se basa en un conjunto de funciones recursivas
- Cada símbolo no terminal genera una función
- En esta función se selecciona la regla de producción a ejecutar en función del valor del siguiente token
- La pila se sustituye implícitamente por la pila de llamadas

- Ejemplo

Gramática

- <E> → <T> <E'>
- <E'> → + <T> <E'>
- <E'> → λ
- <T> → <F> <T'>
- <T'> → * <F> <T'>
- <T'> → λ
- <F> → num
- <F> → (<E>)

Conjunto de predicción

- Pred(<E> → <T> <E'>) = { num, (}
- Pred(<E'> → + <T> <E'>) = { + }
- Pred(<E'> → λ) = {), $ }
- Pred(<T> → <F> <T'>) = { num, (}
- Pred(<T'> → * <F> <T'>) = { * }
- Pred(<T'> → λ) = { +,), $ }
- Pred(<F> → num) = { num }
- Pred(<F> → (<E>)) = { (}

- Ejemplo

```
class Parser {
  private Token nextToken;
  private AnaLex scanner;

  public boolean parse(InputStream stream) { … }

  private void match( int kind ) throws ParseException { … }
  private void parseE( ) throws ParseException { … }
  private void parseEP() throws ParseException { … }
  private void parseT() throws ParseException { … }
  private void parseTP() throws ParseException { … }
  private void parseF() throws ParseException { … }
}
```

- Ejemplo

```
public boolean parse(InputStream stream) {
  this.scanner = new AnaLex(stream);
  this.nextToken = scanner.getNextToken();
  try {
    parseE();
    if(nextToken.getKind() == Token.EOF) return true;
    else return false;
  } catch(Exception ex) {
    return false;
  }
}
```

- Ejemplo

```
private void match(int kind) throws ParseException {
  if(nextToken.getKind() == kind) {
    nextToken = scanner.getNextToken();
  } else {
    throw new ParseException(nextToken, kind);
  }
}
```

- Ejemplo

```
private void parseE() throws ParseException {
  int[] expected = { Token.NUM, Token.LPAREN };
  switch(nextToken.getKind()) {
    case Token.NUM:
    case Token.LPAREN:
        parseT();
        parseEP();
        break;
    default:
        throw new ParseException(nextToken,expected);
  }
}
```

- Ejemplo

```
private void parseEP() throws ParseException {
  int[] expected = { Token.PLUS, Token.RPAREN, Token.EOF };
  switch(nextToken.getKind()) {
    case Token.PLUS:
      match(Token.PLUS);
      parseT();
      parseEP();
      break;
    case Token.RPAREN:
    case Token.EOF:
      break;
    default:
      throw new ParseException(nextToken,expected);
  }
}
```

- Ejemplo

```
private void parseT() throws ParseException {
  int[] expected = { Token.NUM, Token.LPAREN };
  switch(nextToken.getKind()) {
    case Token.NUM:
    case Token.LPAREN:
      parseF();
      parseTP();
      break;
    default:
      throw new ParseException(nextToken,expected);
  }
}
```

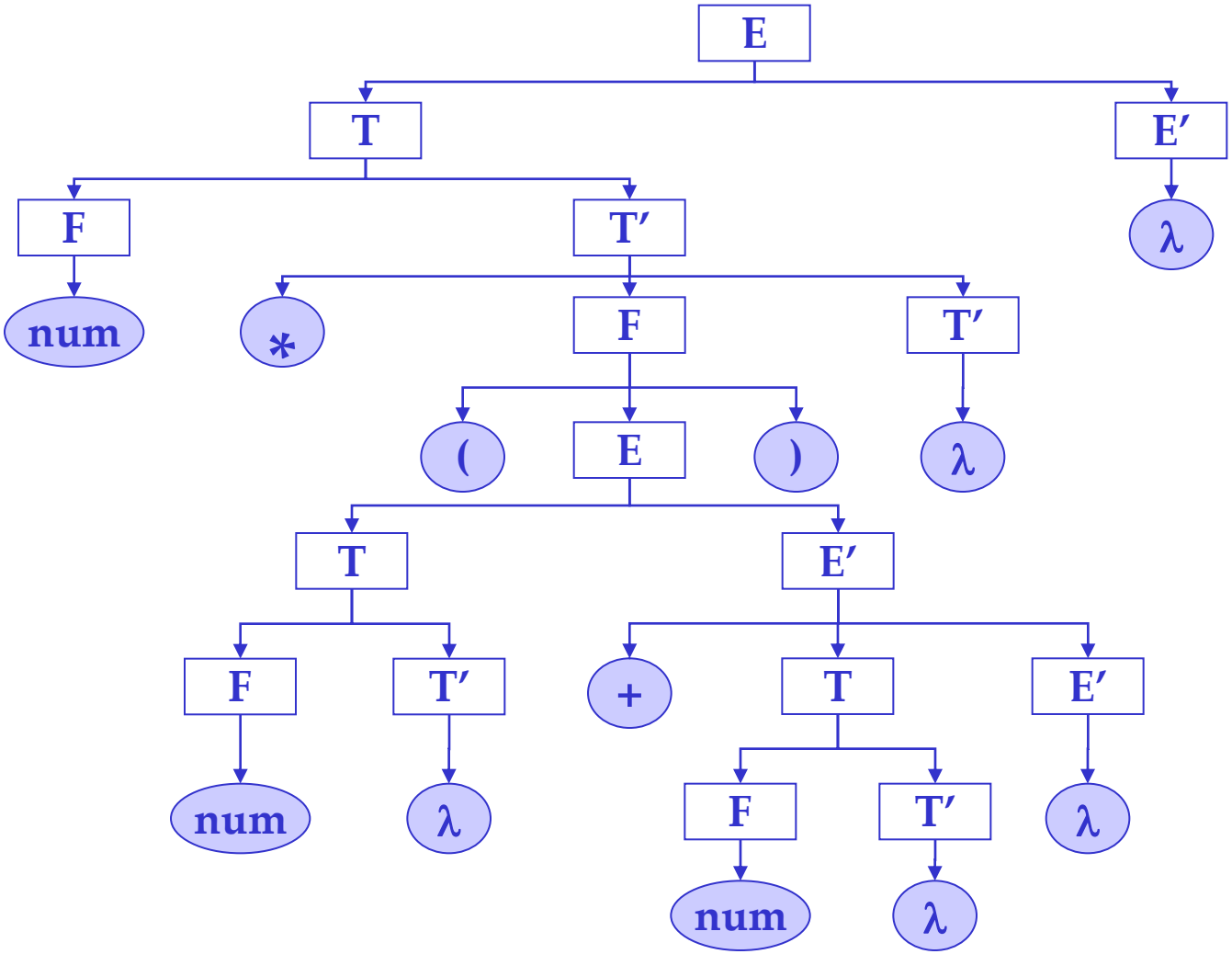
- Ejemplo

```
private void parseTP() throws ParseException {
  int[] expected = { Token.PROD, Token.PLUS,
                     Token.RPAREN, Token.EOF};
  switch(nextToken.getKind()) {
    case Token.PROD:
      match(Token.PROD);
      parseF();
      parseTP();
      break;
    case Token.PLUS:
    case Token.RPAREN:
    case Token.EOF:
      break;
    default:
     throw new ParseException(nextToken,expected);
  }
}
```

- Ejemplo

```
private void parseF() throws ParseException {
  int [] expected = { Token.NUM, Token.LPAREN };
  switch(nextToken.getKind()) {
    case Token.NUM:
      match(Token.NUM);
      break;
    case Token.LPAREN:
      match(Token.LPAREN);
      parseE();
      match(Token.RPAREN);
      break;
    default:
      throw new ParseException(nextToken,expected);
  }
}
```

- Ejemplo: traza de la cadena “num * (num + num)”



3.1 Características del análisis sintáctico

3.2 Gramáticas libres de contexto

3.3 Notación EBNF

3.4 Tipos de análisis sintáctico

3.5 Análisis sintáctico descendente predictivo

3.6 La condición LL(1)

3.7 ASDP dirigido por tabla

3.8 Analizador Descendente Recursivo

3.9 Gestión de errores

- Hay dos tipos de errores
  - Los producidos por función emparejar/consumir
  - Los producidos en las funciones recursivas / en la tabla de símbolos
- Mensajes de error
  - Primeros: Encontrado nextToken cuando se esperaba token
  - Segundos: Encontrado nextToken cuando se esperaba Pred(<A>)
- Recuperación de errores
  - Usando símbolos de sincronismo

- Simbolos de sincronismo
  - Se añade la función skipTo(Token sync) que se salta los tokens de la cadena de entrada hasta llegar al token de sincronismo.
  - Se modifican las funciones recursivas para detectar los errores y saltar a los tokens de sincronismo.
- Ejemplo
  - <Asig> → <Identificador> <Igual> <Expresion> <PuntoYComa>
  - El punto y coma se utiliza como símbolo de sincronismo

- Ejemplo

```
private void parseAsig() throws ParseException {
  int[] expected = { Token.ID };
  switch(nextToken.getKind()) {
    case Token.ID:
      try {
        match(Token.ID);
        match(Token.IGUAL);
        parseExpr();
        match(Token.PYC);
      } catch(ParseException ex) {
        skipTo(Token.PYC);
      }
      break;
   default:
      throw new ParseException(nextToken,expected);
  }
}
```