TEMA 4

ANÁLISIS SINTÁCTICO DESCENDENTE

Bibliografía:

- Aho, A.V., Sethi, R., Ullman, J.D. (1990), *Compiladores: principios, técnicas y herramientas*, Tema 4, páginas: 186-200.
- Louden, K.C. (1997), *Compiler Construction: Principles and Practice*, Tema 4, páginas: 143-193.
- Vivancos, E. (2000), *Compiladores I: una introducción a la fase de análisis*, Tema 3, páginas: 23-48. Especialmente para ejercicios.

Contenido:

1 Análisis descendente: el autómata *predice/concuerda*.

2 Problemas en el análisis descendente: Recursividad a izquierdas y su eliminación. Indeterminismo en las alternativas.

3 Gramáticas LL(1): Los conjuntos de PRIMEROS y SIGUIENTES. La condición LL(1).

4 Analizador sintáctico predictivo recursivo.

5 Analizador sintáctico predictivo dirigido por tabla.

6 Recuperación de errores en los analizadores descendentes: Los conjuntos de predicción y sincronización. Recuperación en modo de pánico.

7 Limitaciones de los métodos descendentes.

4.1 ANÁLISIS DESCENDENTE: EL AUTÓMATA *PREDICE/CONCUERDA*

Especificación de la sintaxis de un lenguaje mediante gramáticas independientes del contexto (GIC). Estas gramáticas permiten recursividad y estructuras anidadas. Por ejemplo: sentencias `if-else` anidadas, paréntesis anidados en expresiones aritméticas, que no pueden ser representadas mediante expresiones regulares.

La recursividad va a implicar:

- Algoritmos de reconocimiento más complejos, que hagan uso de llamadas recursivas o usen explícitamente una pila, la *pila de análisis sintáctico*.
- La estructura de datos usada para representar la sintaxis del lenguaje ha de ser también recursiva (el árbol de análisis sintáctico), en vez de lineal (caso de un vector de caracteres para almacenar los lexemas en el analizador léxico).

Fundamento de los métodos descendentes: autómata predice/concuerda

En cada paso del proceso de derivación de la cadena de entrada se realiza una *predicción* de la posible producción a aplicar y se comprueba si existe una *concordancia* entre el símbolo actual en la entrada con el primer terminal que se puede generar a partir de esa regla de producción, si existe esta concordancia se avanza en la entrada y en el árbol de derivación, en caso contrario se vuelve hacia atrás y se elige una nueva regla de derivación.

Ejemplo Supongamos la siguiente gramática que permite generar expresiones aritméticas:

$$
\begin{aligned}
&\text{E} \rightarrow \text{T E'} \\
&\text{E'} \rightarrow \text{+ T E'} \mid \text{- T E'} \mid \epsilon \\
&\text{T} \rightarrow \text{F T'} \\
&\text{T'} \rightarrow \text{* F T'} \mid \text{/ F T'} \mid \epsilon \\
&\text{F} \rightarrow \text{(E)} \mid \textbf{num} \mid \textbf{id}
\end{aligned}
$$

Para la entrada `num + id * num`.
Crear el árbol de análisis sintáctico...

Nos restringimos al caso de métodos deterministas (no hay vuelta atrás) teniendo en cuenta un sólo símbolo de preanálisis (componente léxico que se está analizando en la cadena de entrada en ese momento), sabemos exactamente en todo momento que producción aplicar. El símbolo de preanálisis se actualiza cada vez que se produce una concordancia.

Los métodos descendentes se caracterizan porque analizan la cadena de componentes léxicos de izquierda a derecha, obtienen la derivación más a la izquierda y el árbol de derivación se construye desde la raíz hasta las hojas.

Problemas de decisión: ¿Qué producción elegir cuando hay varias alternativas?

$$A \rightarrow \alpha \mid \beta \mid \ldots$$

Conjunto PRIMEROS(α): el conjunto de tokens que pueden comenzar cualquier frase derivable de α, $\alpha \in (V_T \cup V_{NT})^*$.

Conjunto SIGUIENTES(A): el conjunto de tokens que pueden aparecer después de un no-terminal A, $A \in V_{NT}$

4.2 PROBLEMAS EN EL ANÁLISIS DESCENDENTE:

- La recursividad a izquierdas da lugar a un bucle infinito de recursión.
- Problemas de indeterminismo cuando varias alternativas en una misma producción comparten el mismo prefijo. No sabríamos cuál elegir. Probaríamos una y si se produce un error haríamos backtracking. Si queremos que el método sea determinista hay que evitarlas.

Existen transformaciones de gramáticas para su la eliminación. Estas transformaciones **no** modifican el lenguaje que se está reconociendo, pero si que cambian el aspecto de la gramática, que es en general menos intuitiva.

4.2.1 Recursividad a izquierdas y su eliminación

Una gramática es *recursiva por la izquierda* si tiene un no-terminal A tal que existe una producción $A \rightarrow A\ \alpha$.
Algoritmo para eliminar recursividad a izquierdas:

- **primer paso:** se agrupan todas las producciones de A en la forma:

$$A \rightarrow A\alpha_1 \mid A\alpha_1 \mid \ \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ \ldots \ \mid \beta_n$$

  donde ninguna β_i comienza con una A.

- **segundo paso:** se sustituyen las producciones de A por:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon$$

Ejemplo: Eliminar la recursividad a izquierdas de

E → E + T | T
T → T * F | F
F → (E) | **num** | **id**

Una gramática recursiva a izquierdas nunca puede ser LL(1). En efecto, asume que encontramos un *token t*, tal que predecimos la producción $A \rightarrow A\beta$, después de la prediccion, al apilar la parte derecha de la producción, el símbolo A estará en la cima de de la pila, y por tanto, se predecirá la misma producción, y tendiramos de nuevo A en la cima, así hasta que el infinito, produciendose un desbordamiento de la pila de análisis sintáctico.

4.2.2 Indeterminismo en las alternativas. Factorización

Cuando dos alternativas para un no-terminal empiezan igual, no se sabe qué alternativa expandir. Por ejemplo:

Stmt → **if** E **then** Stmt **else** Stmt
| **if** E **then** Stmt
| **otras**

Si en la entrada tenemos el token **if** no se sabe cual de las dos alternativas elegir para expandir $Stmt$. La idea es reescribir la producción para retrasar la decisión hasta haber visto de la entrada lo suficiente para poder elegir la opción correcta.

Stmt → **if** E **then** Stmt Stmt' | **otras**
Stmt' → **else** Stmt | ϵ

Algoritmo para factorizar la gramática:

- **primer paso:** para cada no-terminal A buscar el prefijo α más largo común a dos o más alternativas de A, $A \rightarrow \alpha\beta_1|\alpha\beta_2|\ldots|\alpha\beta_n|\gamma$
- **segundo paso:** Si $\alpha \neq \epsilon$, sustituir todas las producciones de A, de la forma $A \rightarrow \alpha\beta_1|\alpha\beta_2|\ldots|\alpha\beta_n|\gamma$, donde γ representa todas las alternativas de A que no comienzan con α por:

$$\text{A} \rightarrow \alpha \text{ A'} \mid \gamma$$
$$\text{A'} \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

4.3 GRAMÁTICAS LL(1)

Las gramáticas LL(1) permiten construir de forma automática un analizador determinista descendente con tan sólo examinar en cada momento el símbolo actual de la cadena de entrada (símbolo de preanálisis) para saber que producción aplicar.

- L: método direccional, procesamos la entrada de izquierda a derecha (*from left to rigth*).
- L: obtenemos derivación más a la izquierda (*left-most derivation*).
- 1: usamos un símbolo de preanálisis para decidir la producción a aplicar.

Teorema 1: Una gramática LL(1) no puede ser recursiva a izquierdas y debe estar factorizada.
Teorema 2: Una gramática LL(1) es no ambigua.

Las afirmaciones inversas no tienen porque ser ciertas.

Ejemplo de gramática LL(1).

$$\text{S} \rightarrow \text{a B}$$
$$\text{B} \rightarrow \text{b} \mid \text{a B b}$$

Para la cadena `aabb`.

Construir el árbol ...

4.3.1 ¿Cómo garantizar que una gramática es LL(1): los conjuntos PRIMEROS y SIGUIENTES?

Sea $\alpha \in (V_N \cup V_T)^*$, PRIMEROS($\alpha$) indica el conjunto de terminales que pueden aparecer al principio de cadenas derivadas de α.

Si $\alpha \stackrel{*}{\Rightarrow} \sigma\sigma_1 \ldots \sigma_n$ entonces $\{\sigma\} \in$ PRIMEROS(α) con $\sigma \in V_T$

Si $\alpha \stackrel{*}{\Rightarrow} \epsilon$ entonces $\{\epsilon\} \in$ PRIMEROS(α)

Sea $A \in V_N$ entonces SIGUIENTES(A) indica el conjunto de terminales que en cualquier momento de la derivación pueden aparecer inmediatamente a la derecha (después de) A.
Sea $\sigma \in V_T$, $\{\sigma\} \in$ SIGUIENTES(A) si existe $\alpha, \beta \in (V_N \cup V_T)^*$, tales que $S \stackrel{*}{\Rightarrow} \alpha A \sigma \beta$.

Algoritmo para calcular el conjunto de PRIMEROS:

Para X un único símbolo de la gramática (terminal o no-terminal). Entonces PRIMEROS(X) consiste de símbolos terminales y se calcula:

- Si X es un terminal o ϵ, entonces PRIMEROS(X)=$\{$X$\}$
- Si X es un no-terminal, entonces para cada producción de la forma $X \rightarrow X_1 X_2 \ldots X_n$, PRIMEROS(X) contiene a PRIMEROS(X_1) $- \{\epsilon\}$. Si además para algún $i < n$ todos los conjuntos PRIMEROS(X_1) ... PRIMEROS(X_i) contienen a ϵ, entonces PRIMEROS(X) contiene a PRIMEROS(X_{i+1}) $- \{\epsilon\}$.

Para α cualquier cadena, $\alpha = X_1 X_2 \ldots X_n$, de terminales y no-terminales, entonces PRIMEROS(α) contiene a
PRIMEROS(X_1) $- \{\epsilon\}$. Para algún $i = 2 \ldots n$, si PRIMEROS(X_k) contiene $\{\epsilon\}$ para todos los $k = 1 \ldots i-1$, entonces PRIMEROS(α) contiene a PRIMEROS(X_i) $- \{\epsilon\}$. Finalmente, si para todo $i = 1 \ldots n$, PRIMEROS(X_i) contiene ϵ, entonces PRIMEROS(α) contiene a $\{\epsilon\}$.

Algoritmo para calcular el conjunto de SIGUIENTES:

Dado $A \in V_{NT}$, entonces SIGUIENTES(A) consiste de terminales y del símbolo $\#$ (símbolo de fin de cadena) y se calcula como:

- Si A es el símbolo de inicio, entonces $\{\#\} \in$ SIGUIENTES(A)
- Si hay una producción $B \rightarrow \alpha A \gamma$, entonces $PRIMEROS(\gamma) - \{\epsilon\} \in SIGUIENTES(A)$
- Si existe una producción $B \rightarrow \alpha A$ ó $B \rightarrow \alpha A \gamma$ tal que $\{\epsilon\} \in PRIMEROS(\gamma)$ entonces SIGUIENTES(B) $\subset$ SIGUIENTES(A).

Ejemplo: calcular PRIMEROS y SIGUIENTES para la gramática

E $\rightarrow$ T E'
E' $\rightarrow$ + T E' | - T E' | ϵ
T $\rightarrow$ F T'
T' $\rightarrow$ * F T' | / F T' | ϵ
F $\rightarrow$ (E) | **num** | **id**

PRIMEROS(E)={(,id,num}
PRIMEROS(T)={(,id,num}
PRIMEROS(F)={(,id,num}
PRIMEROS(E')={+,-,ϵ}
PRIMEROS(T')={*, /, ϵ}

SIGUIENTES(E)={#,)}
SIGUIENTES(E')={#,)}
SIGUIENTES(T)={#,),+,-}
SIGUIENTES(T')={#,),+,-}
SIGUIENTES(F)={+,-,*,/,),#}

4.3.2 La condición LL(1)

Para que una gramática sea LL(1) se debe cumplir que:

1 Para las producciones de la forma $A \rightarrow \alpha_1|\alpha_2|\ldots|\alpha_n$ se debe cumplir que:

$PRIMEROS(\alpha_i) \bigcap PRIMEROS(\alpha_j) = \emptyset$ para todo i,j ($i \neq j$).

Esta regla permite decidir qué alternativa elegir conociendo sólo un símbolo de la entrada.

2 Si $A \in V_N$, tal que $\{\epsilon\} \in$ PRIMEROS(A), entonces:

$PRIMEROS(A) \bigcap SIGUIENTES(A) = \emptyset$

Esta condición garantiza que para aquellos símbolos que pueden derivar la cadena vacía, el primer símbolo que generan y el siguiente que puede aparecer detrás de ellos sean distintos, sino no sabríamos cómo decidir si vamos a empezar a reconocer el no-terminal o que ya lo hemos reconocido.

Comprobar que la gramática anterior para generar expresiones aritméticas es LL(1).

4.4 ANALIZADOR SINTÁCTICO PREDICTIVO RECURSIVO

En este método se considera a cada regla de la gramática como la definición de un procedimiento que reconocerá al no-terminal de la parte izquierda. El lado derecho de la producción especifica la estructura del código para ese procedimiento: los terminales corresponden a concordancias con la entrada, los no-terminales con llamadas a procedimientos y las alternativas a casos condicionales en función de lo que se esté observando en la entrada.

Se asume que hay una variable global que almacena el componente léxico actual (el símbolos de preanálisis), y en el caso en que se produce una concordancia se avanza en la entrada, o en caso contrario se declara un error.

Implementación de un analizador sintáctico recursivo Consta de:

- Un procedimiento, que llamaremos *Parea/Match* que recibe como entrada un terminal y comprueba si el símbolo de preanálisis coincide con ese terminal. Si coinciden se avanza en la entrada, en caso contrario se declara un error sintáctico (lo que esperamos en la gramática no coincide con la entrada).

```
Procedimiento Parea(terminal)
inicio
      si preanalisis == terminal entonces
                preanalisis=siguiente token
      sino
                error sintáctico
fin
```

- Un procedimiento para cada no-terminal con la siguiente estructura:
  - Para las reglas de la forma $A \rightarrow \alpha_1|\alpha_2|\ldots|\alpha_n$, decidir la producción a utilizar en función de los conjuntos PRIMEROS(α_i).

```
Procedimiento A()
inicio
      según preanalisis está en:
                PRIMEROS(α1):  {proceder según alternativa α1 }
                PRIMEROS(α2):  {proceder según alternativa α2 }
                ...
                PRIMEROS(αn):  {proceder según alternativa αn }
      fin_según
      si preanálisis no pertenece a ningún PRIMEROS(αi)
      entonces error sintáctico, excepto si existe la
      alternativa A → ε en cuyo caso no se hace nada
fin
```

  - Para cada alternativa α_i del no-terminal, proceder analizando secuencialmente cada uno de los símbolos que aparece en la parte derecha terminales y no-terminales.

```
Si es un no-terminal haremos una llamada a su procedimiento ;
Si es un terminal haremos una llamada al procedimiento Parea con
ese terminal como parámetro.
```

- Para lanzar el analizador sintáctico se hace una llamada al procedimiento que corresponde con el símbolo de inicio en la gramática, el axioma. ¡ Cuidado, no olvidar hacer una llamada previa al analizador léxico para inicializar el símbolo de preanálisis (variable global) con el primer token del fichero de entrada!.

Implementar un analizador sintáctico predictivo recursivo para: `Procedimiento`

$$
\begin{aligned}
&E \rightarrow T\ E' \\
&E' \rightarrow +\ T\ E' \mid -\ T\ E' \mid \epsilon \\
&T \rightarrow F\ T' \\
&T' \rightarrow *\ F\ T' \mid /\ F\ T' \mid \epsilon \\
&F \rightarrow (\ E\ ) \mid \textbf{num} \mid \textbf{id}
\end{aligned}
$$

```
Expresion()
{
      Termino();
      Expresion_Prima();
}

Procedimiento Expresion_Prima()
{
      según preanálisis está en:
                {TKN_MAS}:
                          Parea(TKN_MAS);
                          Termino();
                          Expresion_Prima();
                {TKN_MENOS}:
                          Parea(TKN_MENOS);
                          Termino();
                          Expresion_Prima();
      si no está en ninguno de los anteriores entonces no hacer nada
      fin_según
}

Procedimiento Termino()
{
      Factor();
      Termino_Prima();
}
```

```
Procedimiento Termino_Prima()
{
      según preanálisis está en:
                {TKN_POR}:
                          Parea(TKN_POR);
                          Factor();
                          Termino_Prima();
                {TKN_DIV}:
                          Parea(TKN_DIV);
                          Factor();
                          Termino_Prima();
      si no está en ninguno de los anteriores entonces no hacer nada
      fin_según
}

Procedimiento Factor()
{
      según preanálisis está en:
                {TKN_ABREPAR}:
                          Parea(TKN_ABREPAR);
                          Expresion();
                          Parea(TKN_CIERRAPAR);
                {TKN_NUM}:  Parea(TKN_NUM);
                {TKN_ID}:  Parea(TKN_ID);
      si no está en ninguno de los anteriores entonces error();
      fin_según
}
```

Se introduce una producción más (se amplia la gramática):

```
Procedimiento S()
{
      Expresion();
      Parea(TKN_#);
}
```

Analizar la entrada `num * num #` y crear el árbol de análisis sintáctico.

Ejemplo 2: Constuir un analizar predictivo recursivo para la siguiente gramática que genera una declaración de tipos en Pascal

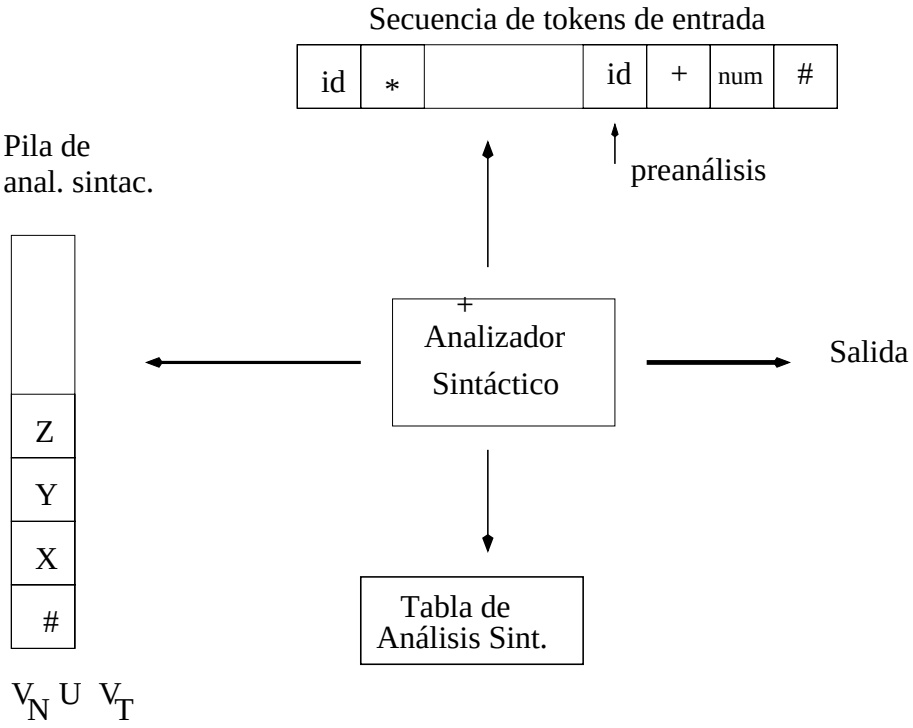
Tipo $\rightarrow$ Simple | $\uparrow$ **id** | **array [** Simple**] of** Tipo
Simple $\rightarrow$ **integer** | **char** | **num puntopunto num**

Construir el árbol para la entrada **array [num .. num] of integer**

4.5 ANALIZADOR SINTÁCTICO PREDICTIVO DIRIGIDO POR TABLA

Se puede construir un analizador sintáctico descendente predictivo no-recursivo sin llamadas recursivas a procedimientos, sino que se utiliza una pila auxiliar de símbolos terminales y no-terminales. Para determinar qué producción debe aplicarse en cada momento se busca en una tabla de análisis sintáctico (*parsing table*) en función del símbolo de preanálisis que se está observando en ese momento y del símbolo en la cima de la pila se decide la acción a realizar.

Esquema de un analizador sintáctico descendente dirigido por tabla



- *Buffer de entrada:* que contiene la cadena de componentes léxicos que se va a analizar seguida del símbolo # de fin de cadena.
- *La pila de análisis sintáctico:* que contiene una secuencia de símbolos de la gramática con el símbolo # en la parte de abajo que indica la base de la pila.

- Una *tabla de análisis sintáctico:* que es una matriz bidimensional $M[X, a]$ con $X \in V_{NT}$ y $a \in V_T \cup \{\#\}$. La tabla de análisis sintáctico dirige el análisis y es lo único que cambia de una analizador a otro (de una gramática a otra).
- *Salida:* la serie de producciones utilizadas en el análisis de la secuencia de entrada.

El analizador sintáctico tiene en cuenta en cada momento, el símbolo de la cima de la pila X y el símbolo en curso de la cadena de entrada a (el símbolo de preanálisis). Estos dos símbolos determinan la acción a realizar, que pueden ser:

- Si $X == a == \#$, el análisis sintáctico ha llegado al final y la cadena ha sido reconocida con éxito.
- Si $X == a \neq \#$, el terminal se desapila (se ha reconocido ese terminal) y se avanza en la entrada obteniendo el siguiente componente léxico.
- Si X es no-terminal, entonces se consulta en la tabla $M[X, a]$. La tabla contiene una producción a aplicar o si está vacía significa un error. Si es una producción de la forma $X \rightarrow Y_1 Y_2 \ldots Y_n$, entonces se meten los símbolos $Y_n \ldots Y_1$ en la pila (notar el orden inverso). Si es un error el analizador llama a una rutina de error.

Algoritmo de análisis sintáctico predictivo recursivo dirigido por tabla:

Entrada: una cadena w, una tabla de anal. sintáct. y una gramática G

Salida: Si $w \in L(G)$, la derivación más a la izquierda de la cadena w, sino una indicación de error

Método:

Como configuración inicial se tiene en el fondo de la pila el símbolo $\#$, el axioma S en la cima y la cadena $w\#$ en el buffer de entrada.

Iniciar preanálisis con el primer símbolo de $w\#$.

repetir

Sea X el símbolo de la cima de la pila

si X es un terminal o $\#$ **entonces**

si $X == preanalisis$ **entonces**

extraer X de la pila y obtener nuevo componente léxico

sino error();

sino

si $M[X, preanalisis] = X \to Y_1 Y_2 \dots Y_n$ **entonces**

extraer X de la pila

meter $Y_n \dots Y_2 Y_1$, con Y_1 en la cima de la pila

sino error();

hasta_que $X == \#$ (* la pila está vacía y hemos procesado toda la entrada*)

Cadena reconocida con éxito

El análisis sintáctico dirigido por tabla es más eficiente en cuanto a tiempo de ejecución y espacio, puesto que usa una pila para almacenar los símbolos a reconocer en vez de llamadas a procedimientos recursivos.

Cómo construir la tabla de análisis sintáctico

Entrada: una gramática G
Salida: la tabla de análisis sintáctico, con una fila para cada no-terminal, una columna para cada terminal y otra para $\#$.
Método:

- Ampliar la gramática con una producción $S' \to S\#$ donde S es el

axioma.

- Para cada producción de la gramática $A \rightarrow \alpha$ hacer:
  - Para cada terminal $a \in PRIMEROS(\alpha)$,
    añadir la producción $A \rightarrow \alpha$ en la casilla M[A,a].
  - Si $\epsilon \in PRIMEROS(\alpha)$,
    añadir $A \rightarrow \alpha$ en la casilla $M(A, b) \forall b \in SIGUIENTES(A)$.
- Las celdas de la tabla que hayan quedado vacías se definen como error.

Pueden aparecer más de una producción en una misma casilla. Esto supondría que estamos en el caso no-determinista en el que tendríamos que probar una producción y si ésta produce un error, probar la otra... Ineficiente. Las gramáticas LL(1) garantizan que sólo tenemos una producción por casilla. Así, el coste del análisis es lineal con el número de componentes léxicos de la entrada.

> *Teorema:* Una gramática es LL(1) si y sólo si el algoritmo de construcción de la tabla de análisis sintáctico no genera ninguna celda con más de una producción, es decir, si para todo par de símbolos X, a tales que $X \in V_N$ y $a \in V_T$, $||M(X, a)|| = 1$.

Ejemplo: para la gramática anterior

E $\rightarrow$ T E'
E' $\rightarrow$ + T E' | - T E' | ϵ
T $\rightarrow$ F T'
T' $\rightarrow$ * F T' | / F T' | ϵ
F $\rightarrow$ (E) | **num** | **id**

	+	-	*	/	(	)	num	id	#
E	err	err	err	err	TE'	err	TE'	TE'	err
E'	$+TE'$	$-TE'$	err	err	err	ϵ	err	err	ϵ
T	err	err	err	err	FT'	err	FT'	FT'	err
T'	ϵ	ϵ	$*FT'$	$/FT'$	err	ϵ	err	err	ϵ
F	err	err	err	err	(E)	err	num	id	err

Insertar el calculo de la tabla.

Insertar el contenido de la pila, la entrada y la producción utilizada para procesar la entrada `num+num*num#`.

4.6 ANÁLISIS SINTÁCTICO Y ACCIONES SEMÁNTICAS

Las acciones semánticas insertadas en las producciones de la gramática se pueden considerar como si fueran símbolos adiciones y que tienen asociado un procedimiento a ejecutar (Símbolos de acciones). Las llamadas a rutinas semánticas no se les pasa paramétros explícitos, los parámetros necesarios se pueden transmitir a traves de una pila semántica. En los analizadores descendentes recursivos, la pila de análisis sintáctico está "escondida" en la pila de llamadas recursivas a los procedimientos. La pila semántica también se puede "esconder", si hacemos que cada rutina semántica devuelva información.

El programa conductor (driver) para gestionar acciones semánticas sería:

Entrada: una cadena w, una tabla de anal. sintáct. y una gramática G

Salida: Si $w \in L(G)$, la derivación más a la izquierda de la cadena w, sino una indicación de error

Método:

Como configuración inicial se tiene en el fondo de la pila el símbolo $\#$, el axioma S en la cima y la cadena $w\#$ en el buffer de entrada.

Iniciar preanálisis con el primer símbolo de $w\#$.

repetir

Sea X el símbolo de la cima de la pila

si X es un terminal o $\#$

si $X == preanalisis$

extraer X de la pila y obtener nuevo componente léxico

sino error();

sino si $M[X, preanalisis] = X \rightarrow Y_1 Y_2 \ldots Y_n$

extraer X de la pila

meter $Y_n \ldots Y_2 Y_1$, con Y_1 en la cima de la pila

sino si X es símbolo de acción

desapilar, llamar a rutina semántica correspondiente

sino

error();

hasta_que $X == \#$ (* la pila está vacía y hemos procesado toda la entrada*)

Cadena reconocida con éxito

4.7 RECUPERACIÓN DE ERRORES SINTÁCTICOS

Funciones de un gestor de errores:

- Determinar si el programa es sintácticamente correcto (reconocedor).
- Proporcionar un mensaje de error significativo.

  parser error: line 10 column 4, found simbol { expected simbol ;

- Reanudar el análisis tan pronto como sea posible. Sino podemos perder demasiada parte de la entrada.

- Evitar errores en cascada, un error genera una secuencia de sucesivos errores espurios. Para evitarlo se debe consumir (ignorar parte de la entrada).
- Realizar una corrección (reparación) del error. El analizador intenta inferir un programa correcto de uno incorrecto. Corrección de mínima distancia (el número de tokens que deben ser insertados, borrados o cambiados debe ser mínimo). Extremadamente complicado "adivinar" la intención del programador. Sólo útil para situaciones sencillas como: "falta ;".

Los métodos de recuperación de errores suelen ser métodos "ad-hoc", en el sentido de que se aplican a lenguajes específicos y a métodos concretos de análisis sintáctico (descendente, ascendente. etc), con muchas situaciones particulares.

Para emitir un mensaje significativo de error se usa el conjunto de símbolos de PREDICCIÓN definidos para cada no-terminal como:
Dada $A \to \alpha$,
si $\epsilon \in PRIM(A)$ **entonces** $PRED(A) = PRIM(A) - \{\epsilon\} \cup SIG(A)$
sino $PRED(A) = PRIM(A)$

Recuperación en modo de pánico: se ignoran símbolos de la entrada hasta encontrar un componente léxico con el cual reanudar el análisis. En el peor caso se podría haber consumido por completo el resto del fichero que queda por analizar (*panic mode*), en cuyo caso no sería mejor que simplemente hacer terminar con el análisis al detectar el error.

Si se implementa con cuidado puede ser mejor que lo que su nombre implica (N. Wirth en su intento de mejorar su imagen le llamó *don't panic mode*).

4.7.1 Recuperación de errores sintácticos en el método descendente predictivo recursivo

Implementación

A cada procedimiento se le proporciona un conjunto de tokens de sincronización (cada no-terminal de la gramática se le asocia un conjunto de sincronización). Si se encuentra un error, el analizador avanza en la entrada (consume tokens), ignorando tokens hasta que encuentra uno del conjunto de sincronización, a partir del cual se continua con el análisis.

¿Qué componentes léxicos nos van a servir para establecer una sincronización?

- El conjunto de SIGUIENTES.
- El conjunto de PRIMEROS, para prevenir el ignorar estructuras internas.

Insertar un ejemplo de por qué es importante poner los dos tipos de conjuntos.

Una forma sencilla de implementar la recuperacion en modo de pánico es al principio de cada procedimiento comprobar si el símbolo de preanálisis pertenece al conjunto de SINC. Si no pertenece al conjunto de SINC, entonces ignoramos parte de la entrada hasta que encontremos un token que pertenezca al conjunto de SINC.

Si el token actual pertenece al conjunto de PRIM entonces ejecutamos el contenido del procedimiento, puesto que hemos encontrado lo “primero” que el genera. Sino (es que pertenece al de SIG), salimos del procedimiento sin ejecutar su cuerpo, pues quiere decir que ya hemos encontrado lo que sigue a ese no-terminal y por tanto, no podemos pretender reconocerlo.

```
Procedimiento A(FIRSTSet, FOLLOWSet) {
while not (preanalisis ∈ (FIRSTSet ∪ FOLLOWSet ∪ #) ){
          error, saltamos simbolo en la entrada;
          scanTo (FIRSTSet ∪ FOLLOWSet);
     }
if (preanalisis ∈ FISRTSet)
     ejecutar cuerpo procedimiento
else
     salir
}
```

4.7.2 Recuperación de errores sintácticos en el método dirigido por tabla

Para cada no-terminal se marcan las casillas correspondientes a los símbolos de sincronización con la palabra $sinc$, siempre que no exista una producción.

Implementación:

Si al buscar en la tabla $M[A, a]$ está en blanco, es decir $a \notin SINC(A)$, avanzamos en la entrada hasta encontrar un terminal b de sincronización. Si $b \in SIG(A)$ se saca el no-terminal A de la pila (desistimos de reconocerlo). Si $b \in PRIM(A)$ no se saca y se aplica la producción correspondiente.

La primera situación corresponde al caso en que se han omitido símbolos y no podemos reconocer lo que genera el no-terminal A y el segundo caso corresponde a que hay un exceso de símbolos que hay que ignorar en la entrada.

Insertar el procesado de la entrada $(num + *num)\#$ para la gramática anterior.

4.8 EJERCICIOS

1 (0.2 ptos) Demuestra que la gramática siguiente no es LL(1).

$$A \rightarrow aAa|\epsilon$$

Construye el pseudocódigo correspondiente para un analizador descendente recursivo como si lo fuera y comenta el problema que se encuentra.

2 (0.2 ptos) Demuestra que la gramática siguiente es LL(1).

$$A \rightarrow (A)A|\epsilon$$

Construye el pseudocódigo correspondiente para un analizador descendente y analiza la entrada $(())$.

3 (0.2 ptos) Dada la siguiente gramática:

Stmt → Assign_stmt | Call_stmt | **other**
Assign_stmt → **id :=** Exp
Call_stmt → **id (** Exp **)**

Escribe el pseudocódigo para analizar de forma descendente recursiva esta gramática. Muestra la traza para una entrada concreta.

4 (0.25 ptos) Dada la siguiente gramática:

Lexp → Atom | List
Atom → **num** | **id**
List → **(** Lexp_seq **)**
Lexp_seq → Lexp **,** Lexp_seq | Lexp

- Factoriza la gramática.
- Construye el conjunto de PRIMEROS y SIGUIENTES para los no-terminales de la gramática resultante.
- Muestra que la gramática generada es LL(1).

- Construye la tabla de análisis sintáctico.
- Muestra el contenido de la pila, de la entrada y las producciones aplicadas para la entrada `(a, (b, (2)), (c))`

5 (0.2 ptos) Considera el siguiente fragmento de gramática que permite generar identificadores como etiquetas:

Stnt $\rightarrow$ Label Unlabeled_stmt
Label $\rightarrow$ **id :**
Label $\rightarrow \epsilon$
Unlabeled_stmt $\rightarrow$ **id :=** Expresion

¿Es LL(1)? Si no, transformala para que lo sea. Implementa un analizador descendente recursivo y dibuja la traza de llamadas recursivas a procedimientos para reconocer la cadena de entrada:

```
etiqueta1 :
a:= 3 + 4;
```

6 (0.25 ptos) Dada la siguiente gramática de una simplificación de las declaraciones en C:

Declaration $\rightarrow$ Type Var_list
Type $\rightarrow$ **int** | **float**
Var_list $\rightarrow$ **identifier ,** Var_list | **identifier**

- Factoriza la gramática.
- Construye el conjunto de PRIMEROS y SIGUIENTES para los no-terminales de la gramática resultante.
- Muestra que la gramática resultante es LL(1).
- Construye la tabla de análisis sintáctico.
- Muestra el contenido de la pila, de la entrada y las producciones elegidas para la entrada `int x,y,z`

- Calcula el conjunto de símbolos de sincronización y aplica el método de recuperación en modo de pánico para la entrada `int x y, z` emitiendo un mensaje de error significativo (calcula el conjunto de PREDICCIÓN).

7 (0.2 ptos) Dada la siguiente gramática que genera una lista de sentencias donde pueden aparecer asignaciones, sentencias condicionales y bloques de sentencias:

L → S | L **;** S
S → **id :=** C | **if (** C **) then** S | **begin** L **end**
C → **id = id** | **id** <> **id** | **id and id**

- Demuestra que no es LL(1).
- Hallar una gramática LL(1) equivalente y construye la tabla de análisis sintáctico.
- Muestra el contenido de la pila, de la entrada y las producciones aplicadas para la entrada `if (id=id) then id:= id and id`

Ideas:

- **Parsons** Mostrar la idea de como se puede evitar el tener que hacer backtracking si dotamos al analizador la posibilidad de mirar un simbolo de look-ahead en la entrada para anticoparse ne la determinación de que terminales son derivables a partir de un cierto símbolo. Mostrar ejemplo pagina 96, para indicar lo que significa el conjunto de PRIMEROS.
- Los analizadores que usan el conjunto de PRIMEROS se les conoce como *Analizadores Predictivos*.
- Mostrar el problema de que cuando existen producciones ϵ, el conjunto de PRIMEROS no es suficiente para determinar la producción a utilizar. No nos dice cuando tenemos que usar $A \rightarrow \epsilon$.
- Insertar dibujo pag 99, en donde se refleja por que el conjunto de SIG de un simbolo tiene que incluir al conjunto de SIG del otro.
- Indicar que para el calculo de SIG una regla consigo misma no debe ser considerada.

- Sobre las condiciones para LL(1): si se viola la primera no sabremos que alternativa coger para un mismo símbolo. Si se viola la segunda, enonces abria algun token en la intersección según el cual no podriamos saber si coger la alternativa correspondiente o serivar en ϵ.
- Una forma no-recursiva del analizador predictivo consiste en un programa simple de control que lee de una tabla. Las ventajas es que el procedimiento de control es general, y si se cambia la gramatica solo se reescribe la tabla.