Tema 6 - Implementación de la sintaxis

Objetivos:

- Escribir gramáticas sintácticas de un lenguaje de programación en notación BNF y BNFA(EBNF)
- Conocer los analizadores sintácticos
- Calcular los símbolos directores de gramáticas sintácticas
- Construir analizadores descendentes predictivos recursivos
- Crear analizadores léxico-sintácticos de forma automática

Tema 6 - Implementación de la sintaxis

Apartados:

1. Especificación de la sintaxis de un lenguaje de programación
2. Analizadores sintácticos
3. Cálculo de los símbolos directores
4. Analizadores descendentes predictivos recursivos
5. Generador automático de un analizador sintáctico - actividad práctica

Tema 6 - Implementación de la sintaxis - notación BNF

- **Especificación sintáctica de un lenguaje**

  Las características sintácticas de un L.P. se pueden especificar mediante una gramática independiente del contexto

  Especificación que puede ser en notación BNF (Backus Naur Form) o EBNF (BNFA)

- **Notación BNF**

  La notación BNF suele usarse para describir sintácticamente lenguajes de programación:

  - ✓ Los símbolos no terminales se representan por cadenas significativas delimitadas por los símbolos < y > que definen una secuencia de símbolos terminales
  - ✓ Los símbolos terminales se denotan por cadenas de símbolos, representativas de sí mismo
  - ✓ El símbolo ::= símbolos separadores de la parte izquierda de la parte derecha se laproducción

Notación EBNF

Para hacer más simple y legible las gramáticas escritas en notación BNF se creó la notación EBNF (o BNFA), que permiten simplificar las representaciones de las definiciones de opcionalidad y repetición de elementos

En dicha notación se emplean los metacaracteres siguientes:

- Cuando una subpalabra de la parte derecha de una producción se puede dar 0 o más veces, dicha palabra se delimita por los símbolos llaves {}
- Cuando una subpalabra de la parte derecha de una producción, se puede dar 0 o 1 vez, dicha palabra se delimita por los símbolos corchetes []
- Cuando una subpalabra de la parte derecha de una producción, se puede dar entre una serie de alternativas, la palabra se delimita por los símbolos paréntesis ()

Ventajas del uso de la anterior notación:

- ✓ Se definen las gramáticas de una forma precisa y fácil de comprender
- ✓ Existen grupos de gramáticas donde se puede construir de forma directa un AS
- ✓ Existen herramientas que a partir de la descripción de la gramática se puede generar código de forma automática de un analizador sintáctico

Ejemplos de gramáticas escritas en notación BNF y EBNF

- Gramática en notación BNF que define la sintaxis de las declaraciones de variables en C

  Ejemplo: `int a, aa; float b, bb;`

<declaraciones>::=<una_declaración><declaraciones> | ε

<una_declaración>::= <tipo> <lista_id_var> ;

<lista_id_var>::= id <mas_id>

<mas_id>::= , id <mas_id> | ε

<tipo>::= int |float |…

- Gramática en notación BNFA que define la sintaxis de las declaraciones de variables en C

<declaraciones>::={<una_declaración>;}

<una_declaración>::= <tipo> id { , id }

<tipo>::= int | float |…

Ejemplos de especificaciones sintácticas:

- Comprobar si la siguiente gramática es ambigua. En caso de serlo, modificarla para que no lo sea sin alterar la semántica que define.

  <valor> ::= <signo> <valor> | cte_car | identificador | cte_ent

  <signo> ::= + | - | ε

- La siguiente sintaxis pretende definir un lenguaje de una lista de cero a infinitos identificadores separados por punto y coma. Justificar si es correcta o no, y en caso de no serlo, corregirla.

  <Lista> ::= identificador ; <Lista> | identificador | ε

Creación de gramáticas sintácticas

Ejemplo 1

- Define en notación BNF la sintaxis de las **llamadas a función** en un determinado lenguaje de programación. Dichas llamadas se caracterizan por estar formadas por el identificador de la llamada seguido de uno o dos parámetros delimitados por los símbolos () y separados por el símbolo ´,´. En caso de tener un sólo parámetro ha de ser obligatoriamente un identificador. Pero si tiene dos parámetros, cada uno de ellos podrá ser indistintamente un identificador, una llamada a otra función o una constante entera

Ejemplo 2

- Crear una gramática sintáctica en notación BNF que defina la **declaración de una tabla** comienza por el nombre, seguido de ':' y la palabra reservada tabla. A continuación se definen las dimensiones de la tabla, al menos una. En primer lugar se definen los límites inferiores separados por comas. A continuación, y después de la palabra reservada hasta, los límites superiores separados por comas. Puede que no coincidan el número de limites inferiores y superiores . Cada límite se indica mediante una constante entera (una constante está formada por uno o más dígitos).
  Ejemplo: x : tabla 0 , 1 hasta 9 , 5 , 6

  Modificarla para que existan igual número de límites inferiores que superiores.
  Ejemplo: x : tabla 0 , 1 , 1 hasta 9 , 5 , 6 (* 3 dimensiones: 0..9, 1..5 y 1..6 *).

Ejemplo 3

- Definir una gramática sintáctica escrita en notación BNF que representa la sintaxis de la **declaración de una *tabla*** .La definición de dicha tabla empieza por la palabra reservada TABLA, seguido de un *índice* delimitado por los símbolos [], seguido de la palabra reservada DE y finalizando por el tipo de los elementos de la misma que se representan por un *identificador*. El índice está delimitado por los corchetes y puede ser: un identificador o dos constantes enteras separadas por el símbolo '..' a las que le puede preceder el símbolo '-' .

Ejemplo 4

- Se desea definir la sintaxis de un pequeño lenguaje para **cálculos matemáticos**. Las características de este lenguaje son la siguientes:

  Únicamente es posible declarar variables y realizar sentencias de cálculo (asignaciones a variables), con la peculiaridad de que no hay ninguna ordenación entre declaraciones y sentencias de cálculo, es decir, pueden estar mezcladas y no hay ninguna obligación de que al principio o al final del fuente deba haber una declaración o un cálculo. Dos ejemplos de fuentes en este lenguaje pueden ser los siguientes:

```
int x                    memoria = 7.5
x = 7                    int x
float y                  x = 7 * memoria
```

Ejemplo 4 (continuación)

- Un fuente puede contener cualquier cantidad tanto de declaraciones como de sentencias de cálculo. Sin embargo, aunque de declaraciones es válido que no exista ninguna, la sintaxis exige que un fuente tenga al menos una sentencia de cálculo.

Se pide:

- Definir la sintaxis de este lenguaje de programación en función de los no terminales <Declaración> y <Sent_cálculo>, cuyas sintaxis no deben definirse. Realizar la defínción primero en BNF estándar y a continuación en BNF extendida o ampliada.
- Suponer que la sintaxis del lenguaje anterior se modifica en las siguientes características:
  - En un fuente todas las declaraciones deben estar al principio del mismo, mientras que todas las sentencias de cálculo deben estar al final.
  - El número de sentencias de cálculo en un fuente ha de ser siempre **mayor** que el número de declaraciones. Al menos debe haber una sentencia de cálculo y es válido que no haya ninguna declaración.
  - Todas las sentencias, ya sean de declaración de variables o de cálculo, van separadas por un punto y coma. Por tanto, la última sentencia de cálculo no lleva punto y coma .

Ejemplo 5

- Representar una gramática sintáctica en notación BNF los siguientes lenguajes de sumas de números naturales definidos sobre el alfabeto Σ = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, + }.
- L_0 lenguaje de sumas de números , donde el operador + es binario y no puede anidarse
  - ✓ Ejemplos de palabras del lenguaje: 2+67+9870 ó 35
  - ✓ Ejemplos de palabras no válidas: 34+ ó 34++23
- L_1 lenguaje de sumas en el que los números naturales no pueden tener ceros a la izquierda. El 0 se representará por un único cero, nunca por una secuencia de ceros.
  - ✓ Ejemplo:10+0
  - ✓ Ejemplo de palabra no válida: 03+12
- L_2 lenguaje de sumas en el que los números naturales pueden empezar por 0, aunque no sea el número 0, pero en cada suma se cumple siempre la condición de que el primer y el último número tienen siempre la misma longitud (al menos habrá dos números).
  - ✓ Ejemplo: 10+1+01
  - ✓ Ejemplos de palabras no válidas: 03+129 , 1
- L_3 el lenguaje de sumas en el que los números pueden empezar por 0, aunque no sea el número 0, y se cumple la condición de que al menos dos números tienen la misma longitud.
  Ejemplo: 0503+**30**+450+**89**+203+03

Ejemplo de separación léxico/sintáctica en una gramática

- Sea la siguiete gramática léxico/sintáctica, separar la gramática léxica de la sintáctica

<Tipo> ::= <TipoSimple> | **^** <Id>
| **TABLA '['** <TipoSimple> **']' DE** <Tipo>
<TipoSimple> ::= **ENTEROS** | **CARACTERES** | <Num> **..** <Num>
<Id> ::= <Letra> **{** <Letra> | <Digito> **}**
<Num> ::= <Digito> **{** <Digito> **}**
<Letra> ::= **a** | · · | **z** | **A** | · · | **Z**
<Digito> ::= **0** | · · | **9**

- **Gramática sintáctica**

<tipo>::= <tipo_simple>
| ↑ id
| TABLA '[' <tipo_simple> ']' DE <tipo>
<tipo_simple> ::= ENTEROS| CARACTERES | num ..num>

- **Gramática léxica**

<Id> ::= <Letra> **{** <Letra> | <Digito> **}**
<Num> ::= <Digito> **{** <Digito> **}**
<Letra> ::= **a** | · · | **z** | **A** | · · | **Z**
<Digito> ::= **0** | · · | **9**

- Sea la gramática léxico/sintáctica, separar la gramática léxica de la sintáctica

<Tabla> ::= TABLA '[' <Indice> ']' OF <Tipo>
<Indice> ::= <Limite> .. <Limite> | identificador
<Limite> ::= <Signo> <cte_ent>
<Signo> ::= - | ε
<Tipo> ::=<identificador>
<Identificador> ::= <Letra> <letraDigito>|<letra>
<letraDigito>::=<letra>|<digito>
< cte_ent > ::= <Digito> |cte_ent> <Digito>
<Letra> ::= **a** | · · | **z** | **A** | · · | **Z**
<Digito> ::= **0** | · · | **9**

- **Gramática sintáctica**

<Tabla> ::= TABLA '[' <Indice> ']' OF identificador
<Indice> ::= <Limite> .. <Limite>
| identificador
<Limite> ::= <Signo> cte_ent
<Signo> ::= - | ε

- **Gramática léxica**

<Identificador> ::= <Letra> <letraDigito>|<letra>
<letraDigito>::=<letra>|<digito>
< cte_ent > ::= <Digito> |<cte_ent> <Digito>
<Letra> ::= **a** | · · | **z** | **A** | · · | **Z**
<Digito> ::= **0** | · · | **9**

Ejemplo . Especificación sintáctica

- Definir la declaración de **variables de un lenguaje** L, caracterizadas por lo siguiente forma:
- Comienzan por la palabra reservada VAR seguida de 0 a infinitas listas de variables separadas por punto y coma.
- Una lista de variables, se compone de 1 a infinitos identificadores separados por comas, dos puntos, y finalmente el tipo de las variables, que puede tratarse de uno o dos identificadores.
- Una variable se define por un identificador, cuya forma responde a las siguientes reglas:
- Comienza por una letra, continúa por letras, dígitos o caracteres de subrayado en cuantía indeterminada (0 a infinito), no pueden aparecer dos caracteres de subrayado consecutivos.
  Ejemplo: VAR x, aux_1, aux2_ : entero; i : entero doble
- Determinar el léxico de la declaración de variables de este lenguaje (basta con una enumeración de las palabras que componen el lenguaje). Construir una definición regular para los identificadores.
- Definir la sintaxis mediante una gramática en notación BNFA. No se considerará válida la solución si no se utilizan adecuadamente los metasímbolos que indican repetición y optatividad.
- Definir la sintaxis utilizando la notación BNF estándar.
- Modificar la sintaxis para que la lista de variables puedan declararse simultáneamente variables de diferentes tipos. Por ejemplo, en aux1, aux2, aux3 : entero, real doble, logico, la variable aux1 sería entera, la variable aux2 real doble y la variable aux3 lógica.
- La gramática debe exigir que el número de variables de la parte izquierda sea igual que el número de tipos de la parte derecha.
- Crear una gramática conjunta de las características léxicas y sintácticas

Ejemplo . Especificación sintáctica

La sintaxis de un lenguaje natural es la siguiente:

- Un texto estará formado por un número indeterminado de oraciones separadas por el símbolo ';' o por el símbolo '.', y vendrá terminado en '.'. Al menos habrá una oración.
- Una oración está compuesta de un sujeto y de un predicado.
- Un sujeto es un nombre precedido o no de un artículo, y seguido de uno o más calificativos separados por ','. Un calificativo es un adjetivo.
- El predicado consta de un verbo seguido de un número indeterminado de complementos.
- En primer lugar irá un complemento directo, si existe, después un complemento indirecto, que es optativo, y por último una lista de 0 o más complementos circunstanciales.
- Un complemento directo es una frase nominal.
- La frase nominal es un nombre precedido o no de artículo, y seguido de un calificativo, un calificativo puede ser un adjetivo o predicado.
- Tanto el complemento indirecto como el circunstancial están formados por una preposición seguida de una frase nominal.

a) Indicar cuál es el léxico de dicho lenguaje.

b) Escribir la gramática en notación BNFA que define la sintaxis del lenguaje.

c) Escribir la gramática en notación BNF stantard que especifique la misma sintaxis.

Especificación de las expresiones aritméticas en los lenguajes de programación

- Son importantes en la sintaxis de un lenguaje programación, puesto que forman parte de la mayoría de las estructuras que la puedan forman: asignación, condiciones, control,…
- Partiendo de una gramática de expresión ambigua se hacen las transformaciones que exija el diseño,

<expr>::= <expr> <op> <expr> | <opdo>

<opdo>::= id | cte

<op>::= + |- |* |/

La gramática anterior es ambigua, las palabras se pueden obtener por más de un árbol de derivación, el árbol puede crecer por la derecha o izquierda, quitando la recursión por uno u otro lado la gramática deja de ser ambigua.

Quitando la recursión por la derecha

<expr>::=<expr> <op> <opdo> | <opdo>

<opdo>::= id | cte

<op>::= + |- |* |/

Quitando la recursión por la izquierda

<expr>::=<opdo><op><expr>| <opdo>

<opdo>::= id | cte

<op>::= + |- |* |/

Especificación de las expresiones aritméticas en los lenguajes de programación

- Las anteriores gramáticas tienen el problema de la **precedencia entre operadores**, problema que se puede resolver:

  creando tantos símbolos no terminales como niveles de precedencia existan, asociando el operador menos precedente al símbolo inicial (raiz) y el mas precedente a los operandos (hojas), así <op+> está asociado a <expr> y <op*> a <term>

  <expr>::=<expr> <op+> <term> | <term>
  <term>::=<term> <op*> <opdo> | <opdo>
  <opdo>::= id | cte
  <op*>::= * | / <op+>::= + |-

  <expr>::=<term> <op+> <expr> |<term>
  <term>::=<opdo> <op*> <term> | <opdo>
  <opdo>::= id | cte
  <op*>::= * | / <op+>::= + |-

- La gramática anterior de la izquierda tiene **asociatividad por la izquierda** para los operadores <op+> y <op*> es decir la palabra id+cte+id se evalúa de izquierda a derecha el árbol crece por la izquierda al tener la recursión por la izquierda, mientras que la otra gramática tiene la asociatividad por la derecha para los mismos operadores.

- **Los paréntesis en las expresiones aritméticas**

  El contenido del paréntesis es un nuevo operando y por lo tanto está en el nivel de los operandos. <opdo>::= id | cte | (<expr>)

- **La llamada a una función,**
  Se trata de un operando y como tal está definido en la gramática
  <opdo>::=<llamada>

- **Operadores unarios**

  Tienen mayor precedencia, por lo tanto más cercano a los operandos

  <fact> ::= <unario> <opdo>

  - Las gramáticas siguiente tiene tres niveles de precedencia entre operadores, de menor a mayor: <op+> ,<op*> son binarios y <unario> es unario precede al operando y se puede anidar.
  - Gramática de expresiones con recursión inmediata por la izquierda

  <expr>::= <expr> <op+> <term> | <term>
  <term>::= <term> <op*> <fact> | <fact>
  <fact>::= <unario> <fact> | <opdo>
  <opdo>::= id | cte| <llamada> | (<expr>)
  <op*>::=* |/ <op+>::= + |- <unario>::= -

- Gramática equivalente a la anterior en la cual se ha quitado la recursión inmediata por la izquierda, modificándose la asociatividad de los operadores

<expr>::= <term> <rest_expr>
<rest_expr>::= <op+> <term> <rest_expr> | ε
<term>::= <fact> <rest_term>
<rest_term>::= <op*> <fact> <rest_term> |ε
<opdo>::= id | cte| <llamada> | (<expr>)
<op*>::=* | / <op+>::= + | - <unario>::= -

- Gramática equivalente a la anterior escrita en notación BNFA

<expr> : := <term> { <op_+> <term>
<term> : := <fact> { <op_*> <Factor>}
<fact> : := { - } <opdo>
<opdo> ::= id | cte | <llamada> | (<expr>)
<op*>::= * | / <op+>::= + | -

Sintaxis de un lenguaje de programación (subconjunto de Pascal):

<gramática_sintáctica>::= <parte_decl> <parte_ejec>
<parte_decl>::= <cabecera> <declaraciones>
<cabecera>::= **PROGRAM id ;**
<declaraciones>::= [<decl_cte>] [<decl_tip>] [<decl_var>] [<decl_sub>]
……
<decl_cte>::= **CONST** <una_cte> **;** { <una_cte> **;** }
<una_cte>::= **id =** <constante>
<constante> ::= [<signo>] <cte_numérica> | **literal**
<Signo> ::= + | -
<cte_numérica> ::= **id** | **cte_ent** | **cte_real**
<decl_tip>::=…….
……
<decl_var> ::= **VAR** <una_var> **;**{<una_var> **;}**
<una_var> ::= **id** { **, id** } **:** <tipo>
<tipo> ::= **INTEGER** | **BOOLEAN** | **CHAR** | **REAL**
<decl_sub>::=…….
<parte_ejec>::= **BEGIN** <sent> { **;** <sent> } **END .**
<sent>::= <sent_sel> | <sent_rep> | <sent_comp> | <sent_vacía> |...
<sent_sel>::= <sent_if> | <sent_case> |.....
……
<sent_if>::= **IF** <expr> **THEN** <sent> [<parte_else>]
<parte_else>::= **ELSE** <sent>

En un lenguaje de programación L, se desea definir la sentencia de alternativa múltiple, cuya estructura general es la siguiente :

ALTERNATIVA (expresión) {
alternativa$_1$: grupo sentencias break
.....
alternativa$_n$: grupo sentencias break
POR DEFECTO : grupo sentencias
}

- ✓ *La expresión podrá utilizar identificadores y constantes como operandos, y como operadores los símbolos aritméticos binarios: (+, -, * , /). Los operadores (+ y -) tienen una precedencia menor que los operadores (* y /). Se podrán utilizar paréntesis de la forma habitual para alterar la precedencia. Los operadores + y - pueden ser también operadores unarios (con mayor precedencia).*
- ✓ Puede que no existan alternativas en cuyo caso debe existir la opción POR DEFECTO, que consiste en la palabra reservada POR DEFECTO seguida de un grupo de sentencias. En otro caso de existir alternativas, no es obligatorio que aparezca la opción POR DEFECTO. Si aparece será la última y no llevará break.
- ✓ Cada alternativa está formada por una lista de 1 o más constantes (entera, char) separadas por comas. El grupo de sentencias está formado por 0 o más sentencias separadas por el símbolo ';' , la última de las cuales podrá ser la sentencia break (no es obligatoria pero si aparece será la última).
- ❑ Indicar cuál es el léxico de dicho lenguaje
- ❑ Obtener una gramática en notación BNF y otra en EBNF que especifique la anterior sintaxis.
- Nota: No es necesario la definición de las sentencias, puesto que no se especifica (se dejará en función del no terminal <sentencia>).

Ejercicio de sintaxis:

Una sentencia de asignación comienza con la variable a la que se asigna, que podrá ser el nombre de una variable de tipo simple, o bien referirse a algún componente de una tabla. A continuación el símbolo ':=' y finalmente la expresión que se asigna.
Para referirse a los componentes de una tabla se indicará el nombre seguido entre los delimitadores '[' y ']' de los valores de cada posición separados por comas. Será válido que no aparezca ningún valor, en cuyo caso se refiere a la tabla completa. Cada valor podrá ser exclusivamente una constante entera o un identificador.
Ejemplo: datos [ ] := datos [ ] + resultado [1 , x]
Las expresiones utilizadas en las sentencias de asignación tienen las siguientes características:

- Operandos: constantes enteras y variables, que podrán ser de tipo simple o de componentes de una tabla.
- Operadores:Binarios: '+' (suma), '-' (resta), '*' (producto), '/' (cociente).
- Unarios: '-' (signo), '++'(autoincremento) y '--'(autodecremento)

Todos los operadores binarios tienen asociatividad a la izquierda

Ejercicio de sintaxis continuación:

- **L**a precedencia de los operadores es la siguiente: primero se hacen los operadores unarios, después tienen la misma precedencia los operadores '*' y '/', y finalmente los últimos operadores en realizarse son '+' (suma) y '-' (resta).
- El operador unario '-', de existir, irá siempre delante del operando. Además, se podrá anidar. Por ejemplo, es válido - - - aux y - - - (x + 1)
- Los operadores unarios '++' y '--' podrán ir delante o detrás del operando, que necesariamente será una variable (de tipo simple o de componentes de una tabla). Los demás operandos no pueden llevar este unario. No se pueden anidar ni mezclar con otros unarios en el mismo operando.

  Usos correctos: -- x , x [2] ++ , -- x [ ]

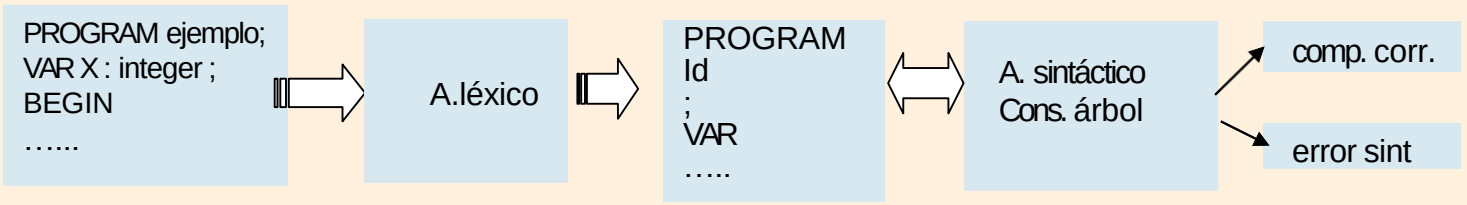
  Usos incorrectos: ++ 3 , -- (aux) , cont * - aux ++ , ++ aux --
- Finalmente, en las expresiones pueden aparecer paréntesis, teniendo en este caso los paréntesis la máxima prioridad. Los paréntesis pueden aparecer anidados.

Ejemplos: x := 7 ; datos [2 , 2] := ++ x * (5 + resto [ ] --) .

Se pide: Teniendo en cuenta que el léxico = { identificador, cte_ent, ,' ', , ':=', '[', ']', '+', '-', '*', '/', '++', -- ' '(', ')'}, definir la sintaxis anterior utilizando la notación BNF .

Analizador sintáctico

El fin de un reconocedor del lenguaje L generado por una gramática G es de reconstruir los árboles de derivación de las palabras de L.



El reconocedor va examinando la palabra de izquierda a derecha, comprobando si el token (pieza) proporcionado por el analizador léxico puede ser generado por la gramática sintáctica que define el lenguaje, en caso de no ser así, informar de los errores de forma precisa e informativa.

Símbolos por adelantado en un proceso de reconocimiento

Símbolos que pueden consultarse para resolver la indeterminación que se presente en el proceso de reconstrucción del árbol de derivación. En el caso de que los símbolos por adelantado no resuelvan el proceso de indeterminación, el análisis resulta no determinista y habrá que realizar un proceso de vuelta atrás.

1 <sintaxis>::= PROGRAM id ; <parte_decl> <parte_ejec> {PROGRAM}

2 <parte_decl>::= <decl_cte> <decl_tip> <decl_var> <decl_sub> { CONST, TYPE,VAR, FUNCTION ,BEGIN}

3 <decl_var>::= VAR <una_var> ; <mas_var> {VAR}

4 | ε { FUNCTION ,BEGIN}

5 <una_var>::= id <mas_id> : <tipo> {id}

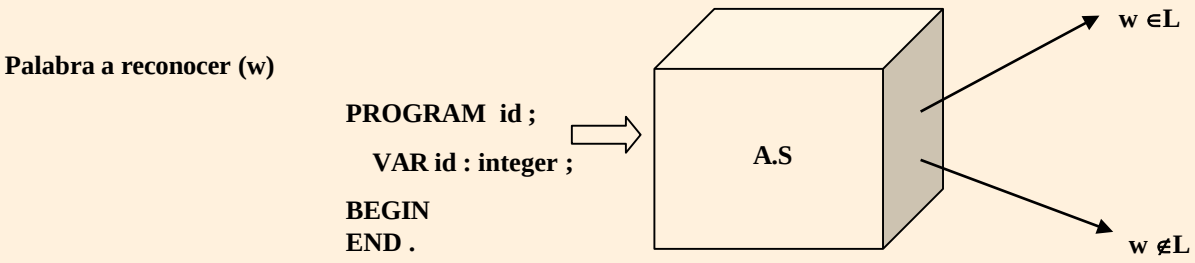
6 <mas_id>::= , ident <mas_id> { , }

7 | ε { : }

8 <tipo>::= integer {integer}

9 | real {real}

10 <parte_ejec>::=BEGIN <list_sent> END. {BEGIN}



1 <sintaxis>::= PROGRAM id ; <parte_decl> <parte_ejec> {PROGRAM}

2 <parte_decl>::= <decl_cte> <decl_tip> <decl_var> <decl_sub> { CONST, TYPE,VAR, FUNCTION ,BEGIN}

3 <decl_var>::= VAR <una_var> ; <mas_var> {VAR}

4 | ε { FUNCTION , BEGIN}

5 <una_var>::= id <mas_id> : <tipo> {id}

6 <mas_id>::= , ident <mas_id> { , }

7 | ε { : }

8 <tipo>::= integer {integer}

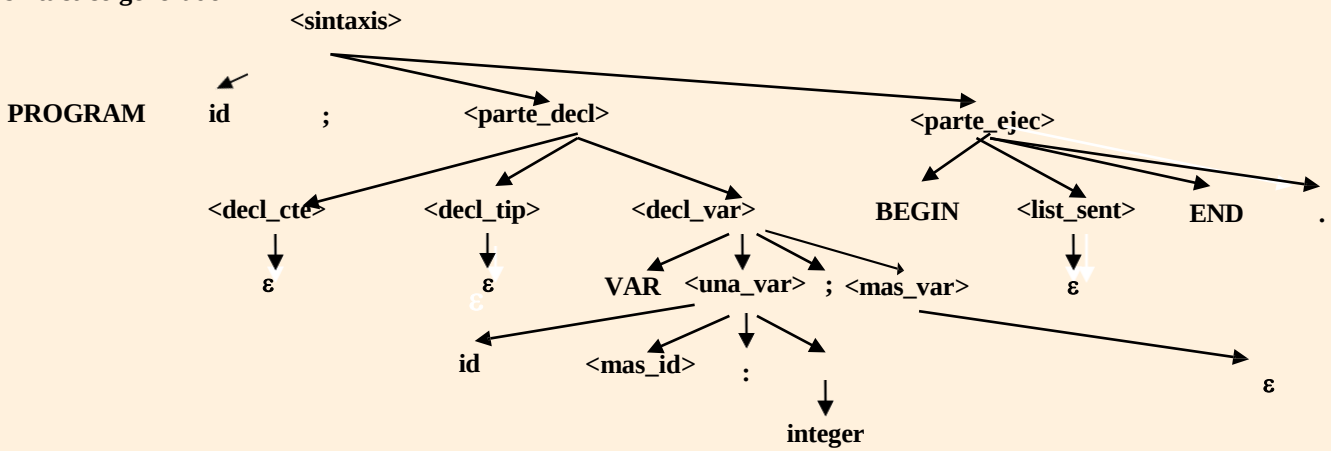
9 | real {real}

10 <parte_ejec>::=BEGIN <list_sent> END. {BEGIN}

Palabra a reconocer (w)

PROGRAM id ;
VAR id : integer ;
BEGIN
END .

Árbol sintáctico generado



Tipos de reconocimiento según la reconstrucción del árbol sintáctico

- Descendentes: tratan de reconstruir el árbol desde la raiz a las hojas.

En este grupo están los **reconocedores LL(K):**

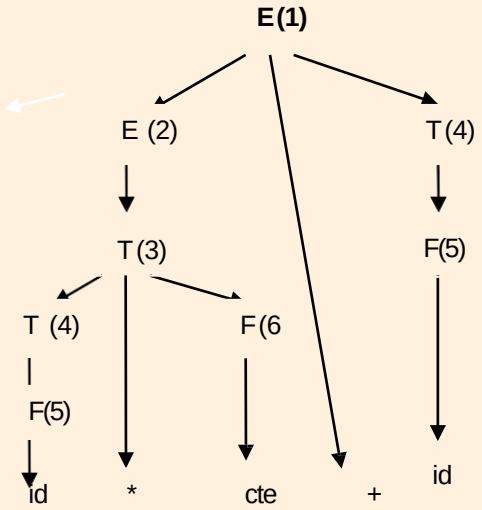
Reconocedores cuya lectura se hace por la izquierda (L), derivación por la izquierda (L) y utilizando (K) símbolos por adelantado para evitar la indeterminación

w= id * cte + id

Ejemplo sencillo

1) E→ E+T |
2) T
3) T→T*F |
4) F
5) F → id |
6) cte |
7) (E)

Reconstrucción descendente
Derivación izquierda
Producciones: 1-2-3-4-5-6-4-5



Reconocedores LL(1) - son los más sencillos en su construcción, pero también son los más limitados sobre el lenguajes a reconocer

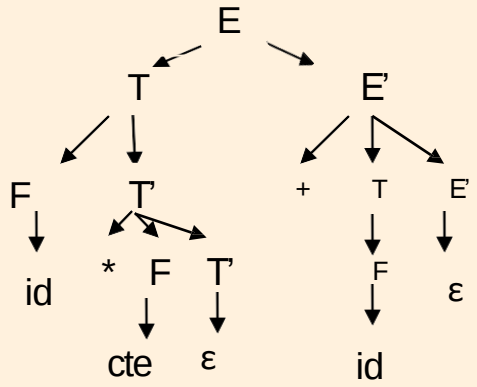
Gramática LL(1) - Una gramática es LL(1), si conociendo un símbolo por adelantado de la entrada éste determinada la producción a derivar, en la creación del árbol de derivación

Lenguaje LL(1) - Un lenguaje es LL(1) si es posible obtener una gramática LL(1) que lo genere.

w= id * cte + id

Gramática LL(1)

$E \rightarrow T\,E'$
$E' \rightarrow +T\,E' \mid \varepsilon$
$T \rightarrow F\,T'$
$T' \rightarrow *F\,T' \mid \varepsilon$
$F \rightarrow cte \mid id \mid (E)$

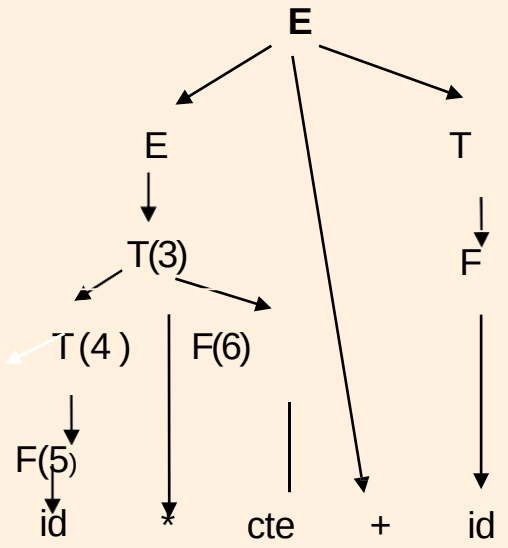


Ascendentes: tratan de reconstruir el árbol desde las hojas a la raíz:

- ✓ En este grupo están los **reconocedores LR (K):**
- ✓ Reconocedores cuya lectura se hace por la izquierda (L), derivación por la derecha (R) y utilizando (K) símbolos por adelantado para evitar la indeterminación.
- ✓ Los reconocedores LR (k) son menos restrictivos en cuanto lenguaje a conocer y más complejos en su construcción

w= id * cte + id

1) E→ E+T |
2) T
3) T→T*F |
4) F
5) F → id |
6) cte |
7) (E)



Reconstrucción ascendente
Derivación derecha
Producciones:
5-4-6-3-2-5-4-1

Palabra anulable $\alpha \in N^*$

- ✓ Es aquella que en un proceso de derivación de cero o más pasos puede hacerse la palabra vacía: $\alpha \rightarrow^* \varepsilon$
- ✓ Los símbolos terminales $a \in \Sigma$ no son anulables
- ✓ ϵ no es anulable.
- ✓ Si $\alpha, \beta \in N^*$ son anulables, entonces $\alpha\beta$ es anulable.

Iniciales de una palabra $\alpha \in (N|\Sigma)^*$

Son todos los símbolos terminales por los que comienzan las palabras que se obtienen de α en un proceso de derivación de cero o más pasos:

$$Ini(\alpha) = \{a \in \Sigma \mid \alpha \rightarrow a\beta,\ \beta \in (N|\Sigma)^*\}$$

Cálculo de los símbolos iniciales

$$Ini(\epsilon) = \emptyset$$

$$Ini(a\beta) = \{a\},\ a \in \Sigma, \beta \in (\Sigma \cup N)^*$$

$$Ini(A\beta) = Ini(A) \text{ si } A \rightarrow^* \varepsilon \text{ no es anulable}$$

$$= Ini(A) \cup Ini(\beta) \text{ si } A \rightarrow^* \varepsilon \text{ es anulable}$$

Seguidores de los símbolos no terminales

- ✓ Símbolos seguidores de un no terminal son todos aquellos símbolos terminales que le siguen en un proceso de derivación

  $Seg(A)$={ a$\in\Sigma$ | S$\rightarrow\alpha$Aaβ, β, $\alpha\in$ (N|Σ)*}

Cálculo de los seguidores de un símbolo no terminal

Se considerarán todas las partes derechas en donde aparece el símbolo no terminal al que se le quiere calcular los seguidores, teniendo en cuenta las siguientes condiciones:

- ✓ Sea la producción $B \rightarrow \alpha A \beta$ $\alpha, \beta \in (N|\Sigma)^*$, entonces:

$$Seg(A) = Ini(\beta) \text{ si } \beta \rightarrow^* \varepsilon \text{ no es anulable}$$

$$Seg(A) = Ini(\beta) \cup Seg(B) \text{ si } \beta \rightarrow^* \varepsilon \text{ es anulable}$$

- ✓ Nota: Si hay que calcular $Seg(S)$, entonces incluir la regla $S' \rightarrow S\$$, donde $S' \in N$.

Símbolos directores de una producción

Se dice que un símbolo terminal es símbolo director de una producción si su lectura por adelantado determina la aplicación de dicha producción

$$Dir\,(A\rightarrow\alpha) = \begin{cases} Ini(\alpha) & \text{si } \alpha\rightarrow^{*}\varepsilon \text{ no es anulable} \\ Ini(\alpha)\cup Seg(A) & \text{si } \alpha\rightarrow^{*}\varepsilon \text{ es anulable} \end{cases}$$

Gramática LL(1) (según los símbolos directores)

Una gramática es LL(1) si:

$$Dir(\alpha_1)\cap Dir(\alpha_2)\cap|\ldots.|\cap Dir(\alpha_n)=\varnothing$$

$$\forall A\rightarrow\alpha_1|\ \alpha_2|\ldots.|\ \alpha_n,\ \alpha_i\in(N|\Sigma)^{*}$$

Ejemplos:

$S\rightarrow aA \quad A\rightarrow Sb|b$ es LL(1)

$S\rightarrow aSb\,|ab$ no es LL(1)

Dadas las siguientes gramáticas en notación formal, comprobar si cumplen la condición LL(1)

S' ---> S$
S ---> abAS | ε
A ---> BA | ε
B ---> a | c

Dir (S ---> abAS) = {a}
Dir (S ---> ε) = Seg (S) = {$}
CUMPLE LL(1)
Dir (A ---> BA) = Inic (BA) = Inic (B) = {a, c}
Dir (A ---> ε) = Seg(A) =
Inic (S) ∪ Seg (S) = {a, $}
NO CUMPLE LL(1)
Dir (B ---> a) = {a}
Dir (B ---> c) = {c}

Dadas las siguientes gramáticas en notación formal, comprobar si cumplen la condición LL(1)

S' ---> S$
S ---> aA | ε
A ---> bB
B ---> aC | cB | ε
C ---> aC | bB | cB | ε

Dir (S ---> aA) = { a }
Dir (S ---> ε) = Seg (S) = { $ }
CUMPLE LL(1)

Dir (B ---> aC) = { a }
Dir (B ---> cB) = { c }
Dir (B ---> ε) = Seg (B) = Seg (A) ∪ Seg (C) = { $ }
Seg (A) = Seg (S) = { $ }
Seg (C) = Seg (B) ∪ Seg (A) = { $ }
CUMPLE LL(1)

Dir (B --->aC) = { a }
Dir (B ---> bB) = { b }
Dir (B ---> cB) = { c }
Dir (B ---> ε) = Seg(B) = { $ }
CUMPLE LL(1)

Propiedades de las gramáticas y condición LL(1)

En ocasiones es posible identificar ciertas propiedades en una gramática de contexto libre que impiden directamente que cumpla la condición LL(1).

Ambigüedad

Una gramática ambigua tampoco puede ser LL(1). La condición LL(1) exige un analizador que reconstruye el árbol de derivación de forma determinista. Por otra parte, la ambigüedad implica que existen dos árboles de derivación para una misma entrada. Intuitivamente esto significa que, en algún momento, el analizador no podrá trabajar de forma determinista ante esa entrada.

Existencia de prefijos comunes

Una gramática tiene prefijos comunes cuando existen al menos dos producciones que, con el mismo no terminal en su parte izquierda, contienen en la parte derecha un prefijo coincidente.

Ejemplo:

$A \rightarrow \delta\alpha \mid \delta\beta$.

Tenemos que:
$Dir(A \rightarrow \delta\alpha) \supset Inic(\delta\alpha) \supset Inic(\delta)$
$Dir(A \rightarrow \delta\beta) \supset Inic(\delta\beta) \supset Inic(\delta)$,

de modo que $Inic(\delta) \subset Dir(A \rightarrow \delta\alpha) \cap Dir(A \rightarrow \delta\beta)$. Si $Inic(\delta) \neq \phi$ es evidente que la gramática no puede ser LL(1).

Eliminar prefijos comunes

Dos o más producciones tienen prefijos comunes cuando teniendo el mismo símbolo no terminal en su parte izquierda, su parte derecha comienza por la misma subpalabra, es decir,

$A \rightarrow \delta\alpha_1 \mid ... \mid \delta\alpha_n \mid \beta_1 \mid ... \mid \beta_m$, $\quad \delta \in (\Sigma \cup N)^+$, $n \geq 2$, $m \geq 0$, $\alpha_i, \beta_j \in (\Sigma \cup N)^*$.

Para eliminar el prefijo común δ se sustituyen las producciones anteriores por

$A \rightarrow \delta X \mid \beta_1 \mid ... \mid \beta_m$
$X \rightarrow \alpha_1 \mid ... \mid \alpha_n$

siendo X un nuevo símbolo no terminal.

Existencia de recursividad inmediata por la izquierda

Una gramática contiene recursividad inmediata por la izquierda si existe al menos una regla de la forma $A \rightarrow A\alpha$.

También debe existir al menos una regla $A \rightarrow \beta$ que no sea recursiva inmediata por la izquierda, ya que en caso contrario A sería una variable inútil, que podría ser eliminada de la gramática.

Considerando la regla $A \rightarrow A\alpha \mid \beta$, tenemos que:

$Dir(A \rightarrow \beta) \supset Inic(\beta)$,
$Dir(A \rightarrow A\alpha) \supset Inic(A\alpha) \supset Inic(A)$.

Pero $Inic(A) = Inic(A\alpha) \cup Inic(\beta)$ y, por tanto, $Inic(\beta) \subset Dir(A \rightarrow A\alpha) \cap Dir(A \rightarrow \beta)$. En consecuencia, la gramática no puede ser LL(1) si $Inic(\beta) \neq \phi$.

Eliminar recursión inmediata por la izquierda

Una producción es recursiva si es de la forma A → αAβ y es recursiva inmediata por la izquierda si tiene la forma A → Aα, donde α, β ∈ (Σ ∪ N)+. Supongamos que tenemos las producciones

$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$

donde las producciones $A \rightarrow \beta_i$ no son recursivas inmediatas por la izquierda.

Las sustituciones:
$A \rightarrow \beta_1 X \mid \ldots \mid \beta_m X$
$X \rightarrow \alpha_1 X \mid \ldots \mid \alpha_n X \mid \epsilon$

o bien, si no se permiten producciones vacías,
$A \rightarrow \beta_1 X \mid \ldots \mid \beta_m X \mid \beta_1 \mid \ldots \mid \beta_m$
$X \rightarrow \alpha_1 X \mid \ldots \mid \alpha_n X \mid \alpha_1 \mid \ldots \mid \alpha_n$

siendo X una nueva variable, eliminan la recursividad inmediata por la izquierda.

Equivalencia entre un reconocedor sintáctico y el reconocimiento de un AP

Apilar ($símbolo\_inicial\_gramática$)
Leer ($símbolo\_entrada$)
SI $cima\_pila$ es símbolo terminal ENTONCES
  SI $cima\_pila$ ES IGUAL A $símbolo\_entrada$
  ENTONCES
    Desapilar y leer entrada
  SINO
    ERROR
SINO
  Desapilar
  buscar parte derecha y apilar parte derecha
HASTA final de la entrada y pila vacía

Una gramática es LL(1) o cumple la condición LL(1), si es posible obtener un AP, que conociendo un símbolo por adelantado, evita el no determinismo