PARSING

Regex convertido en robot

Ing. Max Cerna

Agenda

1. Parsers
2. Gramáticas libres de contexto (CFG)
3. Derivaciones
4. Ambigüedad

Parsers

PARSING

Lenguajes regulares

- Los lenguajes formales más débiles que se pueden utilizar
  - Son capaces de describir patrones simples como palabras clave, identificadores, y ciertas estructuras repetitivas en el código fuente

- Muchas aplicaciones
  - Validación de Patrones Simples
  - Filtros y Procesadores de Texto

PARSING

Muchos lenguajes no son regulares

Las cadenas de paréntesis balanceados no son regulares

$$(^i)^i | i \geq 0$$

¿Qué pueden expresar los lenguajes regulares?

- Expresan propiedades que dependen de contar hasta cierto punto
- Tienen una capacidad limitada de "memoria"
- Debido a esta limitación, no pueden manejar lenguajes que requieran un conteo exacto de elementos

PARSING

INPUT

Secuencia de tokens del lexer

OUTPUT

Árbol de parsing del programa

PARSING


Cadena de caracteres
LEXER
Cadena de tokens
PARSER
Árbol de parsing

PARSING

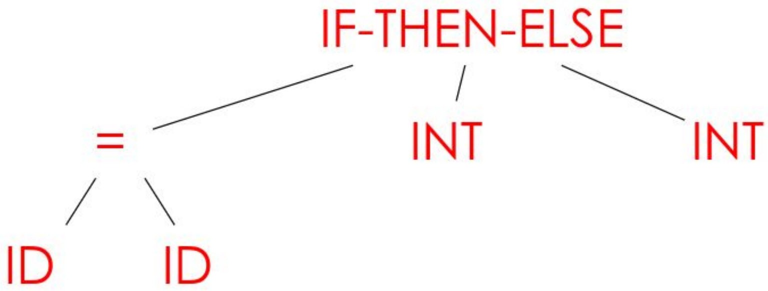
Cool (pseudo lenguaje)

```
if x = y then 1 else 2 fi
```

Entrada al parser (cadena de tokens)

```
IF ID = ID THEN INT ELSE INT FI
```

Salida del parser



Gramáticas libres de contexto (CFG)

CFG

- No todas las cadenas de tokens son programas. . .

- . . . el analizador debe distinguir entre cadenas de tokens válidas e inválidas

- Nosotros necesitamos
  - Un lenguaje para describir cadenas válidas de tokens
  - Un método para distinguir cadenas de tokens válidas de las no válidas

CFG

- Los lenguajes de programación tienen estructura recursiva
- Una EXPR es

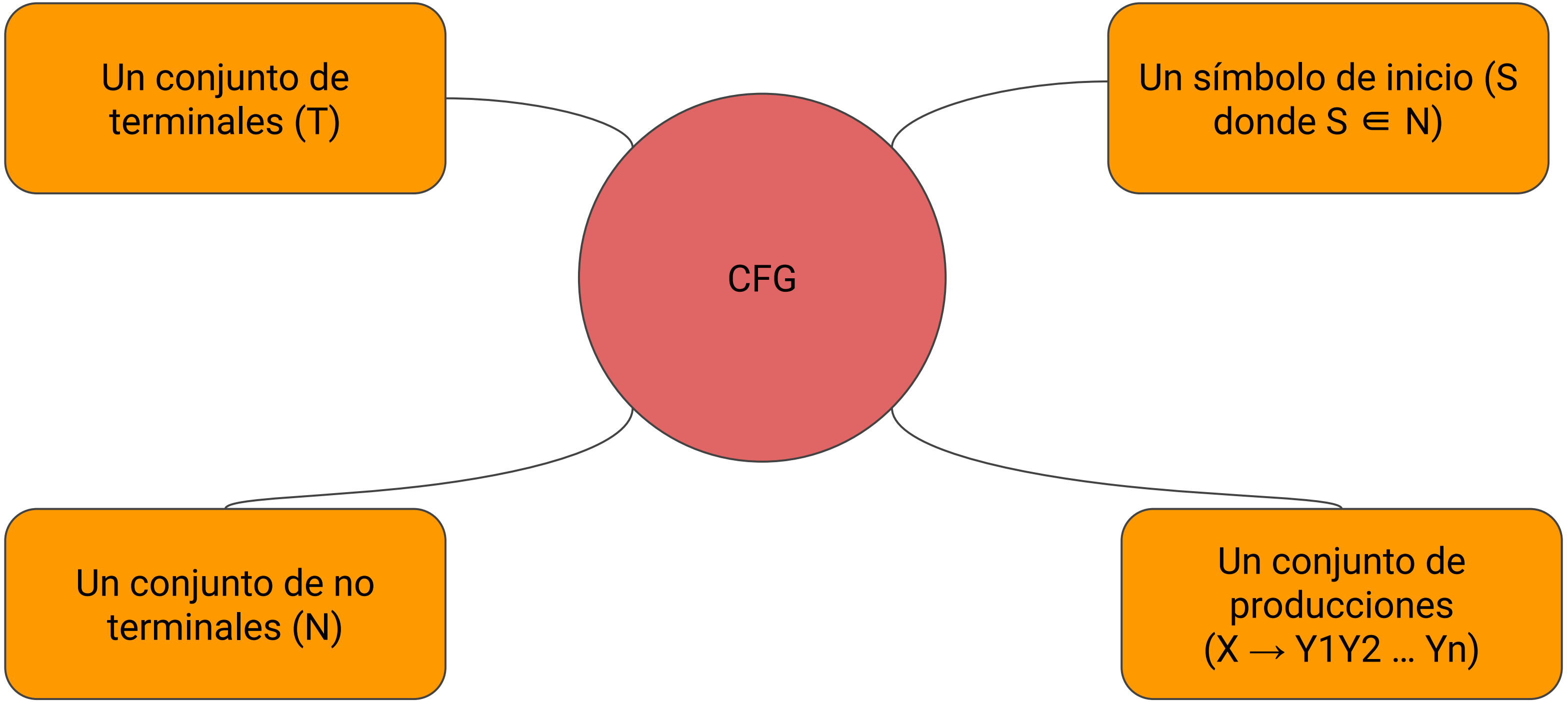
  IF EXPR then EXPR else EXPR fi

  while EXPR loop EXPR pool

  …

- Las gramáticas libres de contexto son una notación natural para esta estructura recursiva

CFG consta de..



CFG

Las producciones se pueden leer como reglas

CFG - Notación

- Los no terminales se escriben en mayúsculas.
- Los terminales se escriben en minúsculas.
- El símbolo de inicio es el lado izquierdo de la primera producción.

CFG - Algoritmo

1) Comience con una cadena con solo el símbolo de inicio S

2) Reemplace cualquier X no terminal en la cadena por el lado derecho de alguna producción ej: X → Y1Yn

3) Repita (2) hasta que no haya no terminales

CFG

- Los terminales se llaman así porque no hay reglas para reemplazarlos
- Una vez generados, los terminales son permanentes
- Los terminales deben ser tokens del idioma.

CFG - EXPRESIONES ARITMÉTICAS SIMPLES

$$
\begin{array}{lcl}
E & \rightarrow & E * E \\
  & | & E + E \\
  & | & (E) \\
  & | & id
\end{array}
$$

CFG - Ejemplo

Cual de las siguientes cadenas están en el lenguaje dado por la CFG

- abcba
- acca
- aba
- abcbcba

$$
\begin{aligned}
S &\to aXa \\
X &\to \varepsilon \\
&\mid\ bY \\
Y &\to \varepsilon \\
&\mid\ cXc
\end{aligned}
$$

CFG

- Permite determinar si una cadena pertenece al lenguaje definido por la gramática.
- Además de verificar si una cadena pertenece al lenguaje, es esencial generar un árbol de análisis sintáctico (parse tree).
- Debe manejar los errores con gracia.
- Necesita una implementación de CFG (p. ej., Bison, CUP).
- La forma de la gramática es importante
  - Muchas gramáticas generan el mismo lenguaje
  - Las herramientas son sensibles a la gramática.

Derivaciones

Derivaciones

Una derivación es una secuencia de producciones:

$S \to \ldots \to \ldots \to \ldots \to \ldots \to \ldots$

Una derivación puede dibujarse como un árbol

- El símbolo de inicio es la raíz del árbol
- Para una producción $X \to Y1...Yn$ agregar los hijos $Y1...Yn$ al nodo

Derivaciones

Gramática `E → E + E   |E * E   |(E)   |id`

Cadena `id * id + id`

Derivaciones

$$E$$

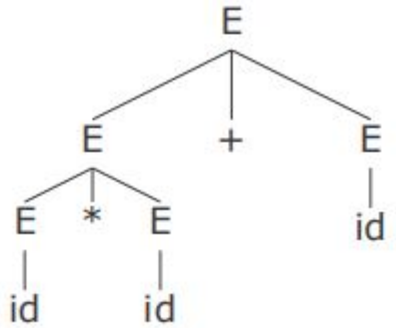
$$\longrightarrow E + E$$

$$\longrightarrow E * E + E$$

$$\longrightarrow id * E + E$$

$$\longrightarrow id * id + E$$

$$\longrightarrow id * id + id$$



Derivaciones

- Un árbol de análisis tiene
  - Terminales en las hojas
  - No terminales en los nodos interiores
- Un recorrido en orden de las hojas es la entrada original
- El árbol de análisis muestra la asociación de operaciones, la cadena de entrada no

Derivaciones

El ejemplo es una derivación por la izquierda (left-most derivation).

En cada paso, reemplaza el no terminal más a la izquierda.

Existe una noción equivalente de una derivación por la derecha (right-most derivation).

$$
\begin{aligned}
& E \\
\longrightarrow\ & E + E \\
\longrightarrow\ & E + id \\
\longrightarrow\ & E * E + id \\
\longrightarrow\ & E * id + id \\
\longrightarrow\ & id * id + id
\end{aligned}
$$

Derivaciones

$$E$$

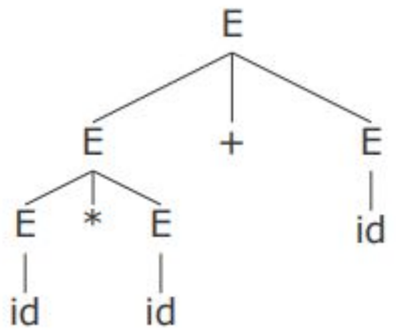
$$\longrightarrow E + E$$

$$\longrightarrow E + id$$

$$\longrightarrow E * E + id$$

$$\longrightarrow E * id + id$$

$$\longrightarrow id * id + id$$



Derivaciones

Tenga en cuenta que las derivaciones por la la derecha y por la izquierda tienen el mismo árbol de análisis

Derivaciones

¿Cuál de las siguientes es una derivación válida de la gramática dada?

```
S → aXa

X → ε | bY

Y → ε | cXc | d
```

1) S
   aXa
   abYa
   acXca
   acca

2) S
   aa

3) S
   aXa
   abYa
   abcXca
   abcbYca
   abcbdca

4) S
   aXa
   abYa
   abcXcda
   abccda

Derivaciones

- No solo estamos interesados en si $s \in L(G)$. Necesitamos un árbol de análisis para s
- Una derivación define un árbol de análisis. Pero un árbol de análisis puede tener muchas derivaciones
- Las derivaciones más a la izquierda y más a la derecha son importantes en la implementación del analizador

Ambigüedad

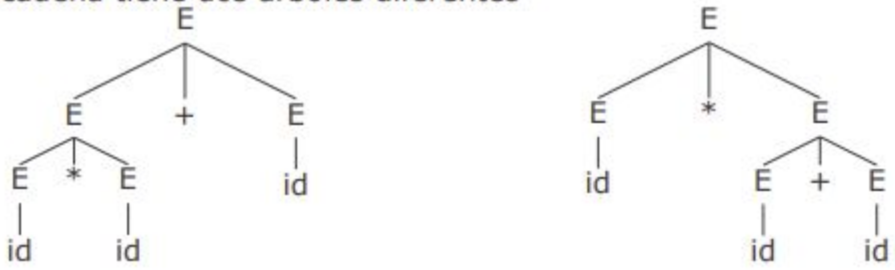
Ambigüedad

Gramática `E → E + E  |E * E  |(E)  |id`

Cadena `id * id + id`

Ambigüedad

La cadena tiene dos árboles diferentes



Ambigüedad

- Una gramática es ambigua si tiene más de un árbol de análisis para alguna cadena.
  - De manera equivalente, existe más de una derivación más a la derecha o más a la izquierda para alguna cadena.
- La ambigüedad es MALA
  - Deja indefinido el significado de algunos programas.

Ambigüedad

¿Cuáles de las siguientes gramáticas son ambiguas?

- $S \rightarrow SS \mid a \mid b$
- $E \rightarrow E + E \mid id$
- $S \rightarrow Sa \mid Sb$
- $E \rightarrow E \mid E + E$
- $E \rightarrow -E \mid id$

Ambigüedad

- Hay varias formas de manejar la ambigüedad
- El método más directo es reescribir la gramática inequívocamente, es decir sin ambigüedad

$E \rightarrow E + E | E$

$E \rightarrow id * E | id | (E) * E | (E)$

Precedencia de $*$ sobre $+$

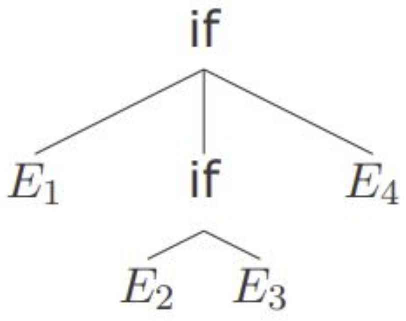
Ambigüedad

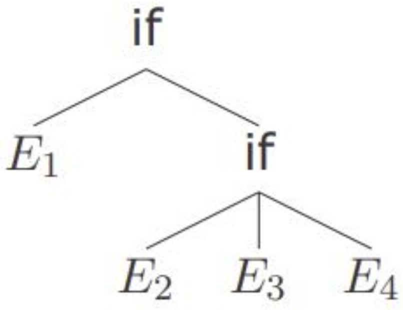
Considere la gramática

```
E →  if E then E
    | if E then E else E
    | OTHER
```

Ambigüedad

La expresión if E_1 then if E_2 then E_3 else E_4 tiene dos árboles





Ambigüedad

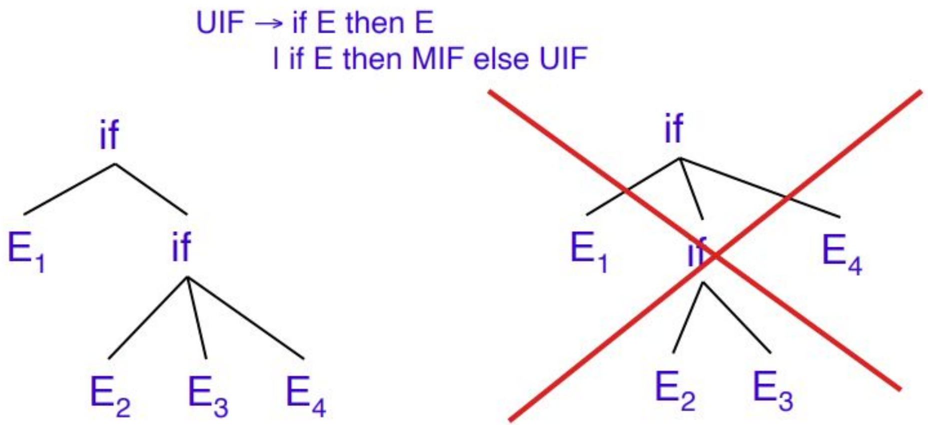
else coincide con el *then* aún sin coincidir más cercano:

```
E →  MIF
   | UIF
MIF →    if E then MIF else MIF
       | OTHER

UIF →    if E then E
       | if E then MIF else UIF
```

Ambigüedad

Entonces expresión `if E1 then if E2 then E3 else E4`



Ambigüedad

- No hay técnicas generales para manejar la ambigüedad
- Es imposible convertir automáticamente una gramática ambigua en una no ambigua
- Usada con cuidado, la ambigüedad puede simplificar la gramática
    - A veces permite definiciones más naturales
    - Necesitamos mecanismos de desambiguación

Ambigüedad

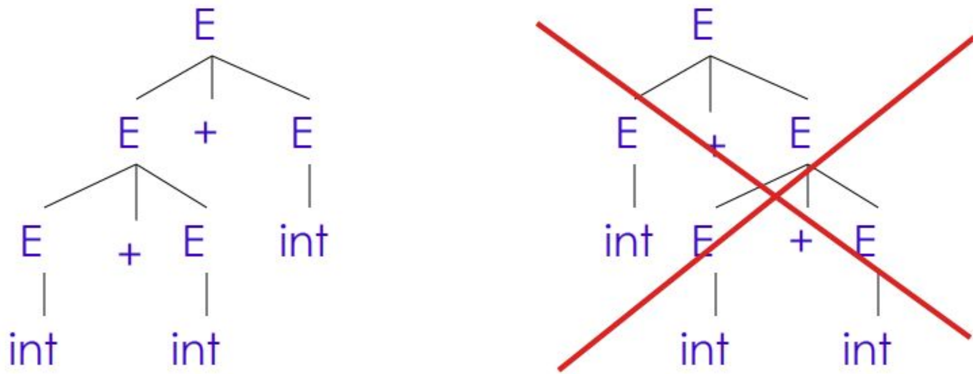
- En lugar de reescribir la gramática:
  - Utiliza la gramática más natural (ambigua)
  - Junto con declaraciones de desambiguación

- La mayoría de las herramientas permiten declaraciones de precedencia y asociatividad para desambiguar las gramáticas

Ambigüedad

Considere la gramática `E → E + E|int`

Dos árboles ambiguos para `int + int + int:`

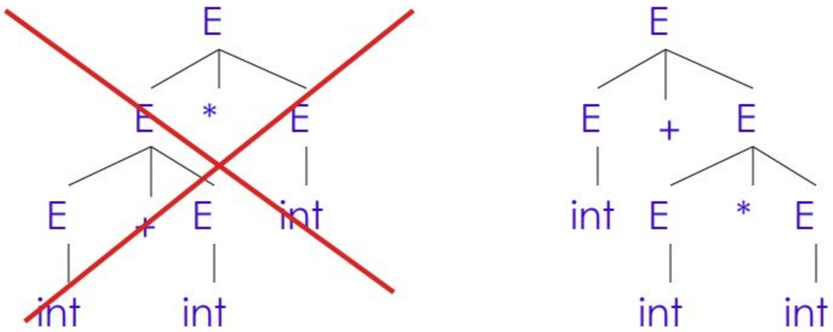


Asociación y declaración por la izquierda: `%left +`

Ambigüedad

Considere la gramática `E → E + E|E * E | int`

Dos árboles ambiguos para `int + int * int:`



Asociación de precedencia: `%left +`

`%left *`

Traducción dirigida por sintaxis

Ing. Max Cerna

Agenda

1. Gestión de errores
2. AST
3. Recursive Descent Parsing
4. Recursive Descent Algorithm
5. Recursión por la izquierda

Gestión de errores

GESTIÓN DE ERRORES

- El propósito del compilador es
  - Para detectar programas no válidos
  - Traducir los válidos
- Muchos tipos de posibles errores (por ejemplo, en C)

Tipo	Ejemplo	Detectado por
Léxico	$	Lexer
Sintáctico	x * %	Parser
Semántico	int x; y = x(3);	Type Checker
Exactitud	tu prox. proyecto de compi	Usuario

GESTIÓN DE ERRORES

- El manejador de errores debe:
  - Reportar los errores de manera precisa y clara.
  - Recuperarse rápidamente de un error.
  - No ralentizar la compilación de código válido.
- Un buen manejo de errores no es fácil de lograr.

GESTIÓN DE ERRORES

- Enfoques de simples a complejos:
  - Modo pánico
  - Producción de errores
  - Corrección automática local o global
- No todos son compatibles con todos los generadores de analizadores.

GESTIÓN DE ERRORES

Modo pánico

El modo de pánico es el método más simple y popular

Cuando se detecta un error:

- Descartar tokens hasta que se encuentre uno con un rol claro
- Continuar desde allí

Buscando tokens de sincronización

- Por lo general, los terminadores de declaraciones o expresiones

GESTIÓN DE ERRORES

Modo pánico

Considere la expresión errónea

```
(1 + +2) + 3
```

Recuperación en modo pánico: Saltar al siguiente entero y luego continuar

Bison/Cup: use el error de terminal especial para describir cuanto saltar en la entrada

```
E → int|E + E|(E)|error int|(error)
```

GESTIÓN DE ERRORES

Modo pánico

Ejemplo, entrada:

int 123a = 5;

float x = 3.14.15;

string name = "Hello World;

GESTIÓN DE ERRORES

Modo pánico

Lexer, macros:

letter = [a-zA-Z]

digit = [0-9]

oper = [+-*/]

whitespace = [\t\n\r]

separator = [;()"]

GESTIÓN DE ERRORES

Modo pánico

Lexer, expresiones regulares:

id = letter (digit | letter)*

num = digit+ (.digit+)?

no_recognized =[^a-zA-Z0-9+\-*/ \t\n\r;()"]

malformed = digit+ letter+ digit* | digit* .[^digit]+

GESTIÓN DE ERRORES

Modo pánico

Parser, producciones SIN manejo de errores:

S -> ASSIGN;

ASSIGN -> id = EXPR

EXPR -> EXPR + TERM | EXPR - TERM | TERM

TERM -> TERM * FACTOR | TERM / FACTOR | FACTOR

FACTOR -> (EXPR) | num | id

GESTIÓN DE ERRORES

Modo pánico

Parser, producciones **CON** manejo de errores:

S -> ASSIGN; | *error ';'*

ASSIGN -> id = EXPR | *error '=' EXPR*

EXPR -> EXPR + TERM | EXPR - TERM | TERM | *error ('+' | '-')*

TERM -> TERM * FACTOR | TERM / FACTOR | FACTOR | *error ('*' | '/')*

FACTOR -> (EXPR) | num | id | *error ('(' | ')' | num | id)*

GESTIÓN DE ERRORES

Producciones de error

Especificar errores comunes conocidos en la gramática

Ejemplo:

- Escribe `5 x` en lugar de `5 * x`
- Añadir la producción `E → E E`

Desventaja:

- Complica la gramática

GESTIÓN DE ERRORES

Producciones de error

Otros ejemplos:

expresiones donde falta un paréntesis de cierre, como en ***5 * (3 + 2***

Añadir la producción E → (E

capturar errores donde un operador es repetido innecesariamente, como en 5 ++ 3

Añadir la producción E → E ++ E

GESTIÓN DE ERRORES

Corrección automática local o global

- Idea: encontrar un programa “cercano” correcto
  - Pruebe las inserciones y eliminaciones de tokens
  - Búsqueda exhaustiva

- Desventajas:
  - Difícil de implementar
  - Ralentiza el análisis de los programas correctos
  - “Cercano“ no es necesariamente el programa “previsto“

GESTIÓN DE ERRORES

- Pasado
  - Ciclo de recompilación lento (incluso una vez al día)
  - Encuentra tantos errores en un ciclo como sea posible

- Presente
  - Ciclo de recopilación rápida
  - Los usuarios tienden a corregir un error/ciclo
  - La recuperación de errores complejos es menos convincente

AST
(Abstract Syntax Trees)

AST

Un parser rastrea la derivación de una secuencia de tokens.

Pero el resto del compilador necesita una representación estructural del programa.

Árboles de sintaxis abstracta (Abstract Syntax Trees)

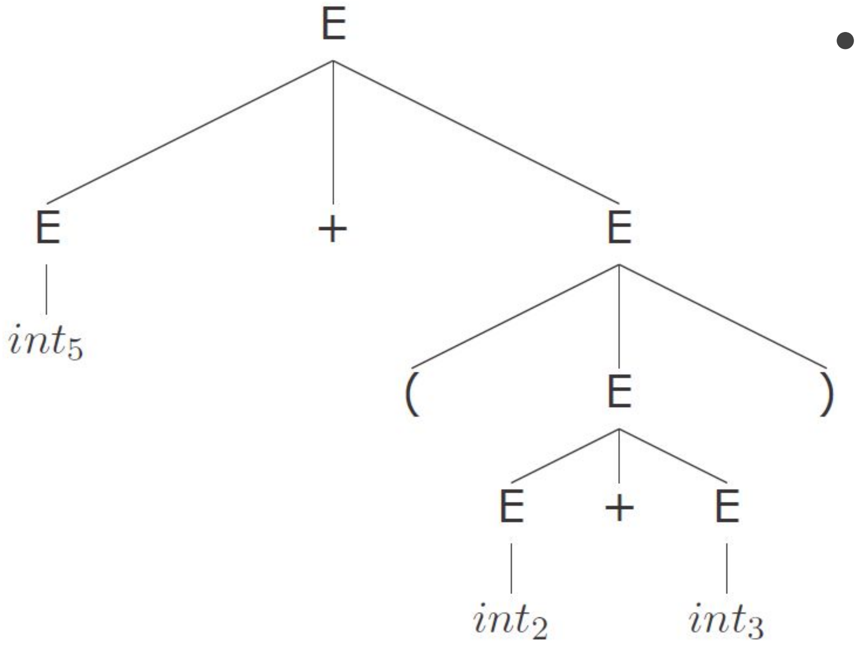
- Como los parse trees vistos hasta ahora pero ignorando ciertos detalles.
- Abreviado como AST

AST

- Considere la gramática
    - `E → int|(E)|E + E`
- Y la cadena
    - `5 + (2 + 3)`
- Después del análisis léxico (una lista de tokens)
    - `int5 + ( int2 + int3 )`

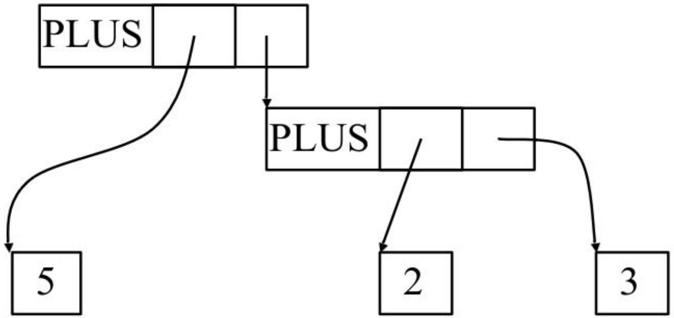
Ejemplo de Parse Tree

Durante el análisis construimos un árbol de análisis



- Un árbol de análisis
  - Rastrea el funcionamiento del parser.
  - Captura la estructura anidada
  - Mucha información (paréntesis, sucesores simples)

Ejemplo de Abstract Syntax Tree

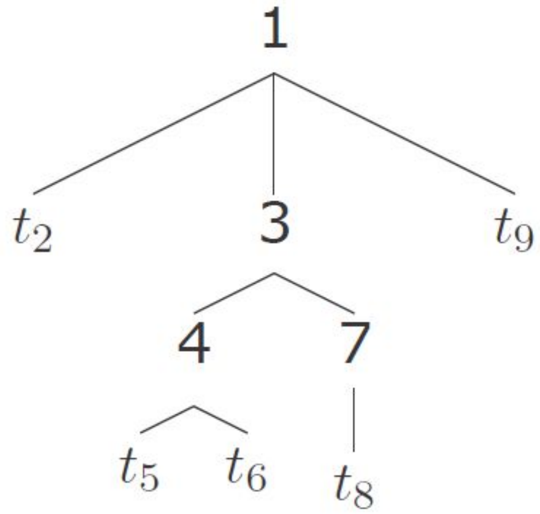


- También captura la estructura de anidamiento
- Pero se abstrae de la sintaxis concreta
    - Más compacto y fácil de usar
- Es una estructura de datos importante en un compilador

Recursive Descent Parsing

RECURSIVE DESCENT PARSING

- El árbol de análisis se construye
  - Desde la parte superior
  - De izquierda a derecha
- Los terminales se ven en orden de aparición en el flujo de tokens: ***t2 t5 t6 t8 t9***



RECURSIVE DESCENT PARSING

- Considere la gramática
  - $E \rightarrow T | T + E$
  - $T \rightarrow int | int * T | (E)$
- El flujo de tokens es: $(int5)$
- Comience con el nivel superior no terminal E
- Pruebe las reglas para E en orden

RECURSIVE DESCENT PARSING

$E \rightarrow T \mid T + E$
$T \rightarrow int \mid int * T \mid (E)$

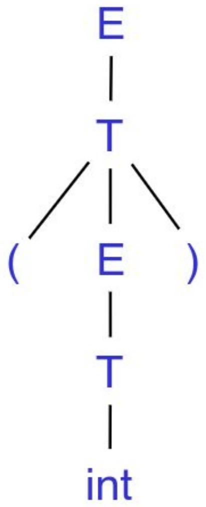
E

(int_5)

RECURSIVE DESCENT PARSING

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$



Aceptado

(int_5)

Recursive Descent Algorithm

Recursive Descent Algorithm (RDA)

Consideraciones a tomar

- TOKEN: Representa un símbolo genérico que puede ser cualquier tipo de token -e.g. INT, OPEN, CLOSE, PLUS, TIMES -

- NEXT: Es un puntero o cursor que apunta al próximo token en la secuencia de entrada que se está analizando.

Recursive Descent Algorithm (RDA)

RDA es un algoritmo que define ***funciones booleanas*** que verifiquen coincidencia de:

1) Un terminal de token dado

```
bool term(TOKEN tok) return *next++ == tok;
```

verifica si el token al que apunta NEXT coincide con el token esperado (tok)

Recursive Descent Algorithm (RDA)

2) La enésima producción de S

   `bool Sn()` …

3) Comprueba todas las producciones de S:

   `bool S()` …

Recursive Descent Algorithm (RDA)

Considerando la gramática:

```
E → T
E → T + E
T → int
T → int * T
T → ( E )
```

Recursive Descent Algorithm (RDA)

Para la producción E → T

```
bool E1() { return T(); }
```

Para la producción E → T + E

```
bool E2() { return T() && term(PLUS) && E(); }
```

Recursive Descent Algorithm (RDA)

Para todas las producciones de E (con backtracking)

```
bool E() {
TOKEN *save = next; //guarda el valor actual del puntero
return (next = save, E1())|| (next = save, E2()); }
```

Recursive Descent Algorithm (RDA)

Funciones para un no terminal T

```
bool T1() { return term(INT); }

bool T2() { return term(INT) && term(TIMES) && T(); }

bool T3() { return term(OPEN) && E() && term(CLOSE); }

bool T() {

    TOKEN *save = next;

    return (next = save, T1()) || (next = save, T2()) ||
    (next = save, T3());

}
```

Recursive Descent Algorithm (RDA)

Para iniciar el analizador

- Inicializar al lado del punto del primer token
  - Invocar E()

- Fácil de implementar a mano

Recursive Descent Algorithm (RDA)

```
bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }
bool E2() { return T() && term(PLUS) && E(); }

bool E() {TOKEN *save = next; return (next = save, E 1())
                                    || (next = save, E2()); }

bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }

bool T() { TOKEN *save = next; return (next = save, T1())
                                    || (next = save, T2())
                                    || (next = save, T3()); }
```

Recursive Descent Algorithm (RDA)

$$E \to E'|E' + id$$
$$E' \to -E'|id|(E)$$

- ☐ Línea 3
- ☐ Línea 5
- ☐ Línea 6
- ☐ Línea 12

```
1  bool term(TOKEN tok) { return *next++ == tok; }

2  bool E_1() { return E'(); }
3  bool E_2() { return E'() && term(PLUS) && term(ID); }
4  bool E() {
5      TOKEN *save = next;
6      return (next = save, E_1()) && (next = save,  E_2());
7  }

8  bool E'_1() { return term(MINUS) && E'(); }
9  bool E'_2() { return term(ID); }
10 bool E'_3() { return term(OPEN) && E() && term(CLOSE); }
11 bool E'() {
12     TOKEN *next = save;  return (next = save, T_1())
13                              || (next = save,  T_2())
14                              || (next = save,  T_3());
15 }
```

Recursión por la izquierda

Recursión por la izquierda

- Considere una producción $S \rightarrow Sa$

```
bool S1() { return S() && term(a); }
bool S() { return S1(); }
```

- S() entra en un ciclo infinito
- Una gramática recursiva por la izquierda tiene una S no terminal $S \rightarrow^{+} S\alpha$ para alguna α
- El descenso recursivo ***no funciona*** en tales casos

Recursión por la izquierda

- Considere la gramática recursiva por la izquierda

  $S \rightarrow S\alpha \mid \beta$

- S genera todas las cadenas que comienzan con β y siguen cualquier número de α

- Se puede reescribir usando la recursividad derecha

  $S \rightarrow \beta S0$

  $S0 \rightarrow \alpha S0 \mid \varepsilon$

Recursión por la izquierda

En general:

$$S \rightarrow S\alpha_1 \mid .. \mid S\alpha n \mid \beta_1 \mid .. \mid \beta_m$$

Todas las cadenas derivadas de S comienzan con uno de $\beta_1, .., \beta_m$ y continúan con varias instancias de $\alpha_1, .., \alpha_n$

Reescribir como

$$S \rightarrow \beta_1 S' \mid .. \mid \beta_m S'$$

$$S' \rightarrow \alpha_1 S' \mid .. \mid \alpha_n S' \mid \varepsilon$$

Recursión por la izquierda

Considere la siguiente gramática:

```
E → E + T
   |T
T → INT
```

lo que puede llevar a un bucle infinito al intentar analizar una expresión como 1 + 2 + 3

Recursión por la izquierda

La gramática:

$$S \rightarrow A\ \alpha\ |\ \delta$$

$$A \rightarrow S\ \beta$$

También es recursiva por la izquierda, dado que si reemplazamos la producción A en S tendremos:

$$S \rightarrow_{+} S\ \beta\ \alpha$$

Recursión por la izquierda

Acerca del Descenso Recursivo

- Estrategia de análisis simple y general
  - La recursividad a la izquierda debe eliminarse primero
  - … pero eso se puede hacer automáticamente
- Históricamente impopular debido al ***backtracking.***
  - Se pensaba que era demasiado ineficiente.
  - En la práctica, con algunos ajustes, es rápida y simple en máquinas modernas.
  - El ***backtracking*** puede ser controlado restringiendo la gramática.

Recursión por la izquierda

PARSERS PREDICTIVOS

Ing. Max Cerna

Agenda

1. Predictive Top-Down Parsers
2. First
3. Follow
4. Tabla LL(1)

Predictive Top-Down Parsers

PREDICTIVE TOP-DOWN PARSERS

- Analizadores predictivos de arriba hacia abajo
- Similares a RDP pero el analizador puede ”predecir” qué, produccion usar
  - Mirando los siguientes tokens
  - Sin retroceso (backtrack)
- Los analizadores predictivos aceptan gramáticas LL(k)
  - L significa escaneo de entrada “de izquierda a derecha”
  - L significa “derivación (más a la/por) izquierda”
  - k significa “predecir basado en k tokens de anticipación”
  - En la práctica, se utiliza LL(1)

PREDICTIVE TOP-DOWN PARSERS

- Por ejemplo, recordemos la gramática

$$
\begin{aligned}
E \rightarrow \; & T + E \\
& \mid T \\
T \rightarrow \; & \texttt{int} \\
& \mid \texttt{int} * T \\
& \mid (E)
\end{aligned}
$$

- Difícil de predecir porque
  - Para T dos producciones comienzan con int
  - Para E no está claro cómo predecir
- Necesitamos factorizar la gramática por/(a la) izquierda

Ejemplo de factorización a la izquierda

Factorizar los prefijos comunes de las producciones

$E \rightarrow T\ X$ // factor común T, lo de la derecha genera nueva producción

$X \rightarrow +\ E \mid \varepsilon$ // nueva producción

$T \rightarrow \mathtt{int}\ Y \mid (\ E\ )$ // factor int, derecha genera nueva producción

$Y \rightarrow *\ T \mid \varepsilon$ // nueva producción

PREDICTIVE TOP-DOWN PARSERS

Elija la alternativa que factoriza correctamente la gramática dada

```
EXPR → if BOOL then { EXPR }
| if BOOL then { EXPR } else { EXPR }
| …
BOOL → true | false
```

1.

```
EXPR → if true then { EXPR }
| if false then { EXPR }
| if true then { EXPR } else { EXPR }
| if false then { EXPR } else { EXPR }
| …
```

2.

```
EXPR→ if BOOL EXPR’
| …
EXPR’ → then { EXPR }
| then { EXPR } else { EXPR }
BOOL → true | false
```

3.

```
EXPR → EXPR’ | EXPR’ else { EXPR }
EXPR’ → if BOOL then { EXPR }
| …
BOOL → true | false
```

4.

```
EXPR → if BOOL then { EXPR } EXPR’
| …
EXPR’ → else { EXPR } | ε
BOOL → true | false
```

PREDICTIVE TOP-DOWN PARSERS

Gramática factorizada

$$E \to TX$$
$$T \to (E)|intY$$
$$X \to +E|\varepsilon$$
$$Y \to *T|\varepsilon$$

siguiente token de entrada

proxima produccion a utilizar

Tabla LL(1)

no terminal

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

PREDICTIVE TOP-DOWN PARSERS

Considere la entrada ***[E, int]***

- "Cuando el no terminal actual es ***E*** y la siguiente entrada es ***int***, usar la producción ***E*** → ***TX***"
- Esto puede generar un ***int*** en la primera posición

	int	*****	**+**	**(**	**)**	**$**
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

PREDICTIVE TOP-DOWN PARSERS

Considere la entrada ***[Y, +]***

- “Cuando el no terminal actual es ***Y*** y la siguiente entrada es **+**, eliminar ***Y***”
- ***Y*** puede ir seguido de **+** solo si ***Y*** → ***ε***

	int	*****	**+**	**(**	**)**	**$**
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

PREDICTIVE TOP-DOWN PARSERS

Considere la entrada ***[E, *]***

- "No hay forma de derivar una cadena que comience con *** desde el no terminal ***E***"

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

PREDICTIVE TOP-DOWN PARSERS

- Método similar al descenso recursivo, excepto
  - Para el S no terminal más a la izquierda
  - Miramos el siguiente token de entrada ***a***
  - Y elige la producción que se muestra en ***[S, a]***
- Una pila registra la frontera del árbol de análisis
  - No terminales que aún no se han ampliado
  - Terminales que aún tienen que coincidir con la entrada
  - Parte superior de la pila = terminal pendiente más a la izquierda o no terminal
- Rechazar al llegar al estado de error
- Aceptar al final de la entrada y pila vacía

LL(1) Parsing Algorithm

```
initialize stack = <S $> and next
repeat
   case stack of
      <X, rest>   : if T[X,*next] = Y1...Yn
                       then stack ← <Y1...Yn, rest>;
                       else  error ();

      <t, rest>   : if t == *next ++
                       then  stack ← <rest>;
                       else error ();

until stack == < >
```

LL(1) Parsing Algorithm

marca el fondo de la pila

Para ***X*** no terminal en la parte superior de la pila, búsqueda de producción

```
initialize stack = <S $> and next
repeat
   case stack of
      <X, rest>   : if T[X,*next] = Y1...Yn
                       then stack ← <Y1...Yn, rest>;
                       else  error ();

      <t, rest>   : if t == *next ++
                       then  stack ← <rest>;
                       else error ();

until stack == < >
```

Pop ***X***, push la producción a la pila. Ten en cuenta que el símbolo más a la izquierda de la producción está en la parte superior de la pila.

Para la terminal ***t*** en la parte superior de la pila, verifique que ***t*** coincida con el siguiente token de entrada.

Ejemplo LL(1) Parsing

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Stack	*Entrada*	*Acción*
E$	int * int$	T X
T X $	int * int$	int Y
int Y X $	int * int$	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε
$	$	ACCEPT

PREDICTIVE TOP-DOWN PARSERS

Considere la siguiente tabla y gramática. ¿Cual es el siguiente estado si el stack actualmente contiene $E'\$$ y la entrada contiene

else { if false then { false } } $?

	if	then	else	{	}	true	false	$
E	if B then { E } E'				ε	B	B	ε
E'			else { E }		ε			ε
B						true	false	

$E \rightarrow$ *if B then {E} E'*$|B|\varepsilon$

$E' \rightarrow$ *else {E}*$|\varepsilon$

$B \rightarrow$ *true*$|$*false*

La Intuición para la construcción de tablas de análisis

- Considere el no terminal A, la producción $A \rightarrow \alpha$ y el token t
- Agregar $T[A, t] = \alpha$ en dos casos

1. Si $A \rightarrow \alpha \rightarrow^* t\beta$
   - α puede derivar t en la primera posición
   - Decimos que $t \in First(\alpha)$
2. Si $A \rightarrow \alpha \rightarrow^* \varepsilon$ y $S \rightarrow^* \gamma A t \delta$
   - Útil si la pila tiene A, la entrada es t y A no puede derivar t
   - En este caso, la única opción es deshacerse de A (derivando ε)
     - Lo anterior solo puede funcionar si t puede seguir a A en al menos una derivación
   - Decimos $t \in Follow(A)$

First

First

Definición:

$First(X) = \{t \mid X \to t\alpha\} \cup \{\varepsilon \mid X \to \varepsilon\}$

Algoritmo:

1) $First(t) = \{t\}$
2) $\varepsilon \in First(X)$
   a) si $X \to \varepsilon$
   b) si $X \to A_1 ... A_n$ y $\varepsilon \in First(A_i)$ para todo $1 \le i \le n$
3) $First(\alpha) \subseteq First(X)$
   a) si $X \to \alpha$
   b) si $X \to A_1 ... A_n\alpha$ y $\varepsilon \in First(A_i)$ para todo $1 \le i \le n$

First

Trabajemos con la gramática factorizada

$E \rightarrow T\ X$

$T \rightarrow (\ E\ )\ |\ int\ Y$

$X \rightarrow +\ E\ |\ \varepsilon$

$Y \rightarrow *\ T\ |\ \varepsilon$

Ejercicio 1 - First

$$
\begin{aligned}
S &\rightarrow A\ B\ C \\
A &\rightarrow x\ A \\
  &\mid\ \varepsilon \\
B &\rightarrow y\ B \\
  &\mid\ \varepsilon \\
C &\rightarrow z
\end{aligned}
$$

Ejercicio 2 - First

$$
\begin{aligned}
S &\rightarrow a\ A \\
&\mid b\ B \\
A &\rightarrow c\ A \\
&\mid d \\
B &\rightarrow \varepsilon
\end{aligned}
$$

Follow

Follow

Definición:

`Follow(X) = {t | S →* βXtδ}`

Razonamiento

- Si **X → AB** entonces **First(B) ⊆ Follow(A)** y **Follow(X) ⊆ Follow(B)**
  - También si **B →* ε** entonces **Follow(X) ⊆ Follow(A)**
- Si **S** es el símbolo inicial entonces **$ ∈ Follow**

Follow

Algoritmo:

1. $\$ \in \text{Follow}(S)$

2. $\text{First}(\beta) - \{\varepsilon\} \subseteq \text{Follow}(X)$
    a. Para cada producción $A \rightarrow \alpha X \beta$

3. $\text{Follow}(A) \subseteq \text{Follow}(X)$
    a. Para cada producción $A \rightarrow \alpha X \beta$ dónde $\varepsilon \in \text{First}(\beta)$

Ejercicio 1 - Follow

$$S \rightarrow X\ Y$$

$$X \rightarrow a\ X \mid \varepsilon$$

$$Y \rightarrow b \mid \varepsilon$$

Ejercicio 2 - Follow

$$S \rightarrow A\ B$$

$$A \rightarrow a\ A \mid \varepsilon$$

$$B \rightarrow b\ B \mid c$$

Tabla LL(1)

Tabla LL(1)

Construir una tabla de análisis T para CFG

Para cada producción $A \rightarrow \alpha$ en G debemos:

- Para cada terminal $t \in$ `First`(α) colocamos $T[A, t] = \alpha$
- Si $\varepsilon \in$ `First`(α), para cada $t \in$ `Follow(A)` colocamos $T[A, t] = \alpha$
- Si $\varepsilon \in$ `First`(α) y $\$ \in$ `Follow(A)` colocamos $T[A, \$] = \alpha$

Tabla LL(1)

Trabajemos con la gramática factorizada

$E \rightarrow T\ X$

$T \rightarrow (\ E\ )\ |\ int\ Y$

$X \rightarrow +\ E\ |\ \varepsilon$

$Y \rightarrow *\ T\ |\ \varepsilon$

Tabla LL(1)

No Terminal	First	Follow
E	{int, (}	{ $,) }
X	{ +, ε }	{ $,) }
T	{ int, (}	{ +, $,) }
Y	{ *, ε }	{ +, $,) }

Ejemplo LL(1) Parsing

$$E \rightarrow T\ X$$
$$T \rightarrow (\ E\ )\ |\ int\ Y$$
$$X \rightarrow +\ E\ |\ \varepsilon$$
$$Y \rightarrow *\ T\ |\ \varepsilon$$

	int	*	+	(	)	$
E	T X			T X		
X						
T						
Y						

No Terminal	First	Follow
E	{int, (}	{ $,) }
X	{ +, ε }	{ $,) }
T	{ int, (}	{ +, $,) }
Y	{ *, ε }	{ +, $,) }

Ejemplo LL(1) Parsing

$$E \rightarrow T\ X$$

$$T \rightarrow (\ E\ )\ |\ int\ Y$$

$$X \rightarrow +\ E\ |\ \varepsilon$$

$$Y \rightarrow *\ T\ |\ \varepsilon$$

	int	*	+	(	)	$
E	T X			T X		
X			+ E			
T						
Y						

No Terminal	First	Follow
E	{int, (}	{ $,) }
X	{ +, ε }	{ $,) }
T	{ int, (}	{ +, $,) }
Y	{ *, ε }	{ +, $,) }

Ejemplo LL(1) Parsing

$E \rightarrow T\ X$

$T \rightarrow (\ E\ )\ |\ \texttt{int}\ Y$

$X \rightarrow +\ E\ |\ \varepsilon$

$Y \rightarrow *\ T\ |\ \varepsilon$

	int	*	+	(	)	$
E	T X			T X		
X			+ E			
T	int Y					
Y						

No Terminal	First	Follow
E	{int, (}	{ $,) }
X	{ +, ε }	{ $,) }
T	{ int, (}	{ +, $,) }
Y	{ *, ε }	{ +, $,) }

Ejemplo LL(1) Parsing

$E \rightarrow T\ X$

$T \rightarrow (\ E\ )\ |\ \mathtt{int}\ Y$

$X \rightarrow +\ E\ |\ \varepsilon$

$Y \rightarrow *\ T\ |\ \varepsilon$

	int	*	+	(	)	$
E	T X			T X		
X			+ E			
T	int Y			(E)		
Y						

No Terminal	First	Follow
E	{int, (}	{ $,) }
X	{ +, ε }	{ $,) }
T	{ int, (}	{ +, $,) }
Y	{ *, ε }	{ +, $,) }

Ejemplo LL(1) Parsing

$E \rightarrow T\ X$

$T \rightarrow (\ E\ )\ |\ int\ Y$

$X \rightarrow +\ E\ |\ \varepsilon$

$Y \rightarrow *\ T\ |\ \varepsilon$

	int	*	+	(	)	$
E	T X			T X		
X			+ E			
T	int Y			(E)		
Y		* T				

No Terminal	First	Follow
E	{int, (}	{ $,) }
X	{ +, ε }	{ $,) }
T	{ int, (}	{ +, $,) }
Y	{ *, ε }	{ +, $,) }

Ejemplo LL(1) Parsing

$E \rightarrow T\ X$

$T \rightarrow (\ E\ )\ |\ int\ Y$

$X \rightarrow +\ E\ |\ \varepsilon$

$Y \rightarrow *\ T\ |\ \varepsilon$

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T				

No Terminal	First	Follow
E	{int, (}	{ $,) }
X	{ +, ε }	{ $,) }
T	{ int, (}	{ +, $,) }
Y	{ *, ε }	{ +, $,) }

Ejemplo LL(1) Parsing

$E \rightarrow T\ X$

$T \rightarrow (\ E\ )\ |\ int\ Y$

$X \rightarrow +\ E\ |\ \varepsilon$

$Y \rightarrow *\ T\ |\ \varepsilon$

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

No Terminal	First	Follow
E	{int, (}	{ $,) }
X	{ +, ε }	{ $,) }
T	{ int, (}	{ +, $,) }
Y	{ *, ε }	{ +, $,) }

Ejercicio 1 - Tabla LL(1)

$S \rightarrow X\ Y$

$X \rightarrow a\ X \mid \varepsilon$

$Y \rightarrow b \mid \varepsilon$

No Terminal	First	Follow
S	{ a, b, ε }	{ $, b }
X	{ a, ε }	{ b }
Y	{ b, ε }	{ $, b }

Ejercicio 1 - Tabla LL(1)

$S \rightarrow X\ Y$

$X \rightarrow a\ X \mid \varepsilon$

$Y \rightarrow b \mid \varepsilon$

	a	b	$
S	XY	XY	XY
X	a X	ε	
Y		b/ε	ε

No Terminal	First	Follow
S	{ a, b, ε }	{ $, b }
X	{ a, ε }	{ b }
Y	{ b, ε }	{ $, b }

Ejercicio 1 - Tabla LL(1)

$S \rightarrow A\ B$

$A \rightarrow a\ A \mid \varepsilon$

$B \rightarrow b\ B \mid c$

No Terminal	First	Follow
S	{ a, b, c }	{ $ }
A	{ a, ε }	{ b, c }
B	{ b, c }	{ $ }

Ejercicio 1 - Tabla LL(1)

$S \rightarrow A\ B$

$A \rightarrow a\ A \mid \varepsilon$

$B \rightarrow b\ B \mid c$

	a	b	c	$
S	A B			A B
A	a A	ε	ε	
B		b B	c	

No Terminal	First	Follow
S	{ a, b, c }	{ $ }
A	{ a, ε }	{ b, c }
B	{ b, c }	{ $ }

Tabla LL(1)

- Si una entrada es definida múltiples veces entonces G no es LL(1)
  - No está factorizada por la izquierda
  - Tiene recursividad por la izquierda
  - Es ambigua

- La mayoría de lenguajes de programación no son LL(1)

Ejemplos y Ejercicios para CFG

Ing. Max Cerna

Recursividad por la izquierda

Si tenemos producciones de la forma:

A-> Aa | b

Estas dos producciones pueden ser sustituidas por:

A-> bA′

A′-> aA′|ε

Ejemplo

Remover recursividad por la izquierda de la siguiente gramatica:

`S → Ab | b`

`A → Ac | Sd | x`

recursividad indirecta de **`S`**, vamos a sustituir la producción **`S`** en **`A → Sd`** usando las producciones de **`S → Ab | b`**

Ahora las producciones de A seran

`A → Ac | Abd | bd | x`

Ejemplo

Introducir una nueva variable para eliminar la recursividad por la izquierda en A, que llamaremos A'.

Dividimos las producciones de A

- Las que comienzan con A (recursivas): **`A → Ac | Abd`**
- Las que no comienzan con A (no recursivas): **`A → bd | x`**

Ejemplo

Las producciones de A se convierten en las partes no recursivas seguidas por A':

A → bdA' | xA'

Las producciones de A' manejarán las partes recursivas:

A' → cA' | bdA' | ε

Ejemplo

Gramática final

$S \rightarrow Ab \mid b$

$A \rightarrow bdA' \mid xA'$

$A' \rightarrow cA' \mid bdA' \mid \varepsilon$

Ejercicio

```
A → Aa | A1 | A2 | b

B → Bb | A

C → Cc | D

D → d | e
```

Ejercicio

¿Cuántas cadenas pertenecen al lenguaje descrito por la siguiente gramática?

S → AB

A → 0 | ε

B → 1B | 1 | ε

Ejercicio

¿Cuántas cadenas pertenecen al lenguaje descrito por la siguiente gramática?

$S \to AB$

$A \to 0 \mid 1 \mid \varepsilon$

$B \to 1 \mid 2 \mid \varepsilon$

Ejercicio

¿Las gramáticas son equivalentes?

Gramática 1: $S \rightarrow aSb \mid \varepsilon$

Gramática 2: $T \rightarrow aTb \mid ab \mid \varepsilon$

Ejercicio

Dada la gramática recursiva por la izquierda:

S → Sa | b

Cuales gramáticas quitan la recursividad por la izquierda y son equivalentes:

a) S → bA
A → aA | ε

b) S → b | bS

c) S → bA
A → a | aA

d) S → A | b
A → aA | a