Teoría de Lenguajes

Intro compiladores + Analizadores Sintácticos Descendentes

Christian G. Cossio Mercado

DC-UBA

1er Cuatrimestre 2017

Compiladores

- Tienen como objetivo convertir un código fuente de programa a otro lenguaje
- En particular importarán aquellos que realizan la conversión a código máquina ejecutable o en código intermedio interpretable (e.g., Java o .NET)
- Adicionalmente, tendrá que poder detectar errores en el código, y poder marcarlos de forma de facilitar su corrección

Compiladores

Proceso de un Compilador

1. **Análisis Léxico**: Se encarga de procesar las cadenas y generar los lexemas correspondientes. Por cada lexema se generará las lista de tokens (par token-valor)
2. **Análisis Sintáctico**: A partir de la salida del analizador léxico se generará el árbol sintáctico correspondiente a la código recibido
3. **Análisis Semántico**: Se genera información para las etapas subsiguientes, y en particular se encargará del chequeo de tipos
4. **Generación de Código Intermedio**
5. **Optimización**
6. **Generación de Código**

El problema de interés

- Dado un lenguaje L definido por la gramática

$$G =< \{E, T, F\}, \{+, *, num, (,)\}, P, E >$$

$$\begin{array}{lcl} E & \rightarrow & E + T \\ E & \rightarrow & T \\ T & \rightarrow & T * F \\ T & \rightarrow & F \\ F & \rightarrow & num \\ F & \rightarrow & (E) \end{array}$$

- y dada la cadena

$$\alpha : (14 + 6) * 2$$

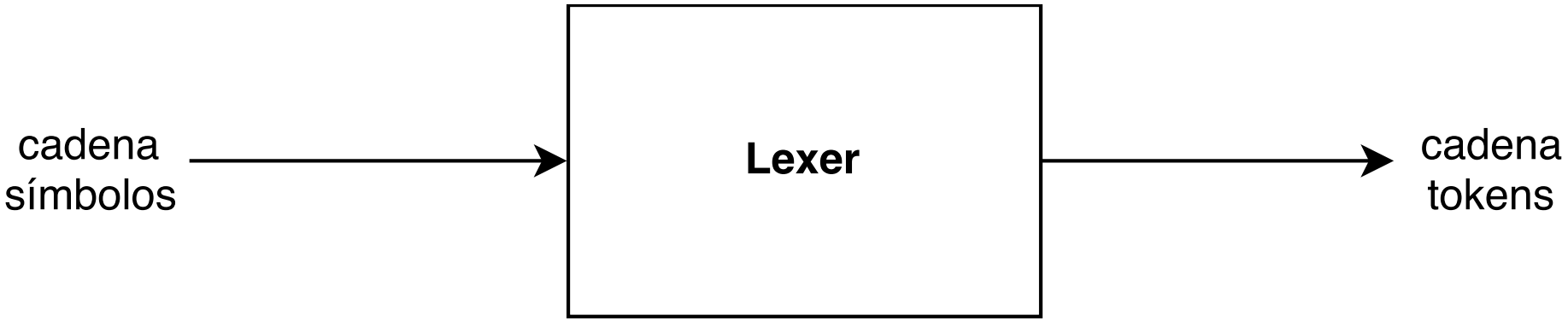
- ¿$\alpha \in L$?

¿Cómo sabe la computadora qué es un **num**?

- ¿Reemplazamos al terminal *num* por los terminales 14, 6 y 2?
- ¿Reemplazamos la producción $F \rightarrow num$ por las producciones $F \rightarrow 14$, $F \rightarrow 6$ y $F \rightarrow 2$?
- ¿Podemos hacer esto para todos los números?
- ¡Son infinitos!

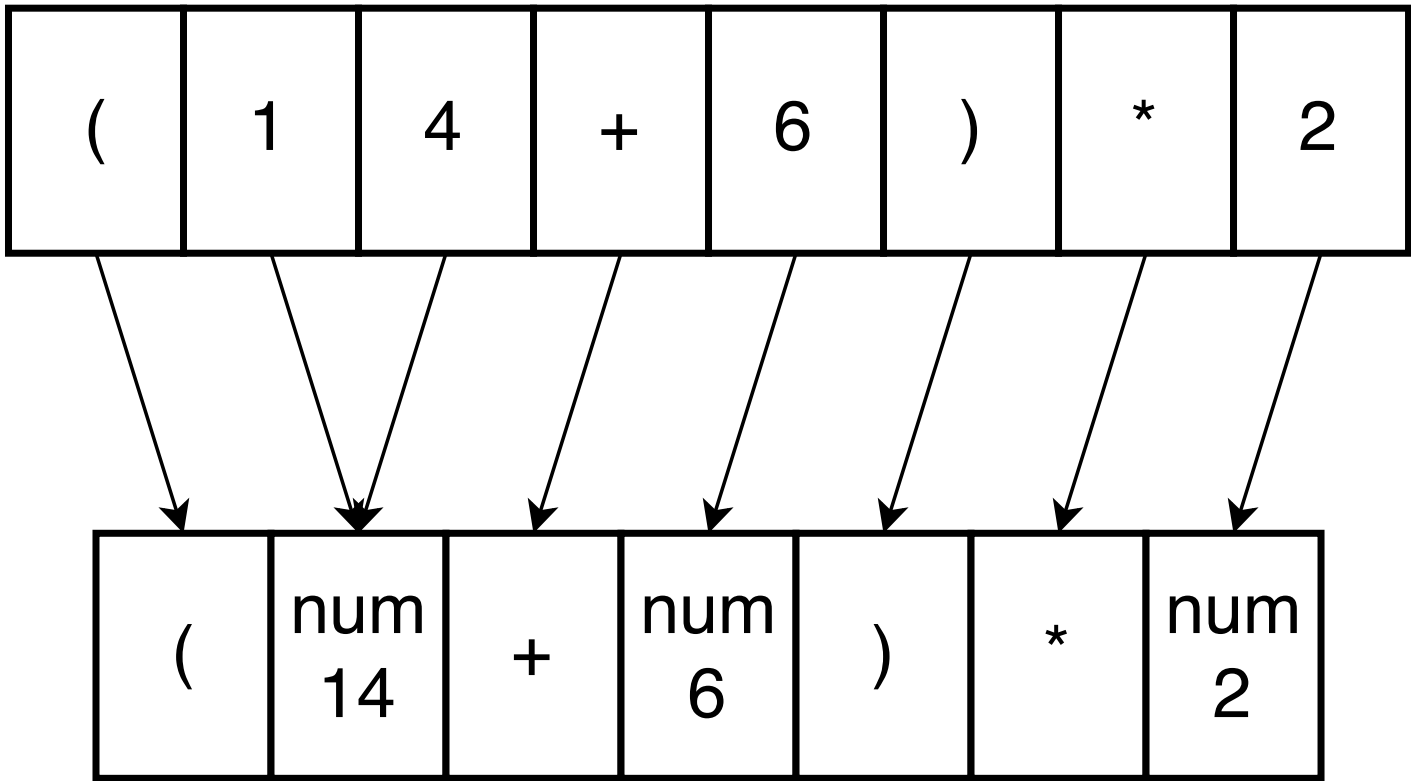
- Ahí es donde entran en juego los analizadores léxicos (lexers)

Analizador Léxico (Lexer)

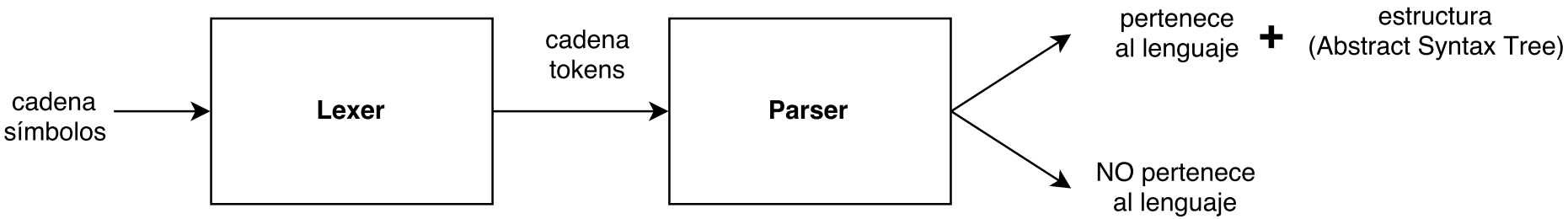


- Usa expresiones regulares
- Agrupa símbolos de la entrada en tokens
- Un token tiene un tipo y un valor

Ejemplo



Lexer + Parser



Analizadores Sintácticos (parsers)

- ¿Para qué sirve un analizador Sintáctico?

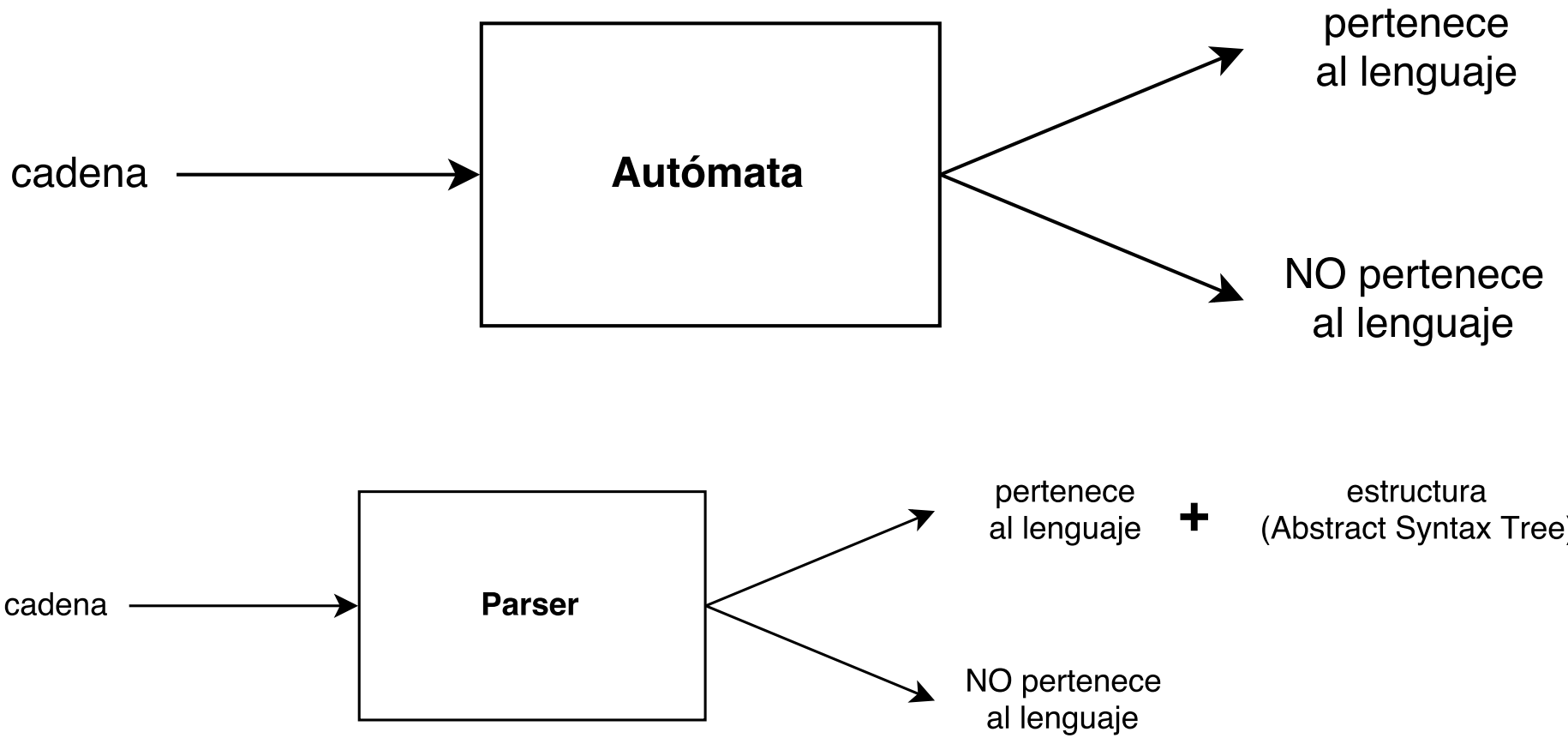
Analizadores Sintácticos (parsers)

- ¿Para qué sirve un analizador Sintáctico?
- En pocas palabras: para verificar si el código sigue la gramática del lenguaje y obtener el árbol sintáctico correspondiente

Analizadores Sintácticos (parsers)

- ¿Para qué sirve un analizador Sintáctico?
- En pocas palabras: para verificar si el código sigue la gramática del lenguaje y obtener el árbol sintáctico correspondiente
- Entonces, ¿para qué querría un analizador sintáctico, si ya tengo a los automátas?

Autómata vs. Parser



Tipos de Analizadores Sintácticos

- Descendentes (top-down):

Tipos de Analizadores Sintácticos

- Descendentes (top-down):
  - Construyen el árbol sintáctico desde la raiz hasta las hojas (la secuencia de tokens).

Tipos de Analizadores Sintácticos

- Descendentes (top-down):
  - Construyen el árbol sintáctico desde la raiz hasta las hojas (la secuencia de tokens).
  - Ejemplos: DFS, parser predictivos recursivos, LL(1), LL(k).

Tipos de Analizadores Sintácticos

- Descendentes (top-down):
  - Construyen el árbol sintáctico desde la raiz hasta las hojas (la secuencia de tokens).
  - Ejemplos: DFS, parser predictivos recursivos, LL(1), LL(k).
- Ascendentes (bottom-up)

Tipos de Analizadores Sintácticos

- Descendentes (top-down):
  - Construyen el árbol sintáctico desde la raiz hasta las hojas (la secuencia de tokens).
  - Ejemplos: DFS, parser predictivos recursivos, LL(1), LL(k).
- Ascendentes (bottom-up)
  - Construyen el árbol sintáctico desde las hojas (la secuencia de tokens) hasta la raiz (el símbolo distinguido).

Tipos de Analizadores Sintácticos

- Descendentes (top-down):
  - Construyen el árbol sintáctico desde la raiz hasta las hojas (la secuencia de tokens).
  - Ejemplos: DFS, parser predictivos recursivos, LL(1), LL(k).
- Ascendentes (bottom-up)
  - Construyen el árbol sintáctico desde las hojas (la secuencia de tokens) hasta la raiz (el símbolo distinguido).
  - Ejemplos: parsers shift-reduce, LR(0), SLR, LALR, LR(K)

Tipos de Analizadores Sintácticos

- Descendentes (top-down):
  - Construyen el árbol sintáctico desde la raiz hasta las hojas (la secuencia de tokens).
  - Ejemplos: DFS, parser predictivos recursivos, LL(1), LL(k).
- Ascendentes (bottom-up)
  - Construyen el árbol sintáctico desde las hojas (la secuencia de tokens) hasta la raiz (el símbolo distinguido).
  - Ejemplos: parsers shift-reduce, LR(0), SLR, LALR, LR(K)
- Otros

Tipos de Analizadores Sintácticos

- Descendentes (top-down):
  - Construyen el árbol sintáctico desde la raiz hasta las hojas (la secuencia de tokens).
  - Ejemplos: DFS, parser predictivos recursivos, LL(1), LL(k).
- Ascendentes (bottom-up)
  - Construyen el árbol sintáctico desde las hojas (la secuencia de tokens) hasta la raiz (el símbolo distinguido).
  - Ejemplos: parsers shift-reduce, LR(0), SLR, LALR, LR(K)
- Otros
  - Generalized LR (GLR), Earley, CYK

Analizadores top-down

- Dada la siguiente gramática:

$$S \rightarrow A \mid B$$
$$A \rightarrow a$$
$$B \rightarrow b$$

- ¿Cómo genero la cadena a?

Analizadores Recursivos descendentes

- Opción A: hacer DFS o BFS

Analizadores Recursivos descendentes

- Opción A: hacer DFS o BFS
  - ¿Qué problemas tiene?

Analizadores Recursivos descendentes

- Opción A: hacer DFS o BFS
  - ¿Qué problemas tiene?
- Opción B: Usar un analizador predictivo

Analizadores Recursivos descendentes

- Opción A: hacer DFS o BFS
  - ¿Qué problemas tiene?
- Opción B: Usar un analizador predictivo
  - Ya tengo un criterio de selección de un regla

Analizadores Recursivos descendentes

- Opción A: hacer DFS o BFS
  - ¿Qué problemas tiene?
- Opción B: Usar un analizador predictivo
  - Ya tengo un criterio de selección de un regla
  - Sigue teniendo algunas debilidades/restricciones

Parsers predictivos descendentes

- Parsers descendentes (parten del símbolo distinguido)
- Utilizan la información del próximo token (i.e., token corriente, o `tc`) y los SD's de cada regla
- Con eso,
  - se crea un procedimiento por cada no terminal, siendo el procedimiento principal (`Main`) el correspondiente al símbolo distinguido
  - dentro de cada procedimiento, se invocan los procedimientos de los no terminales incluidos, se busca los terminales esperados (`match`), o no se hace nada (si es un λ),
  - si hay más de regla para un mismo no terminal, se agregan condicionales para ejecutar sólo la regla que contenga al token actual (tc) en sus SD's
    - se agrega un `else` final con `error()`, para cubrir el caso en que el `tc` no está en los SD de ninguna de las reglas posibles

¿Cómo elegir una regla?

Primeros, Siguiente, Símbolos Directrices

- Vamos a definir dos funciones: Primeros y Siguientes

$$Primeros(\alpha) : (V_N \cup V_T)^* \to \mathbf{P}(V_T)$$
$$Primeros(\alpha) = \{t \in V_T \mid \alpha \stackrel{*}{\Rightarrow} t\beta\}$$

$$Siguientes(N) : V_N \to \mathbf{P}(V_T)$$
$$Siguientes(N) = \{t \in V_T \mid S\$ \stackrel{*}{\Rightarrow} ...Nt...\}$$

- Con eso creamos la función de Símbolos Directrices (SD):

$$SD(A \to \beta) = \begin{cases} \text{Primeros}(\beta) & \text{si } \beta \text{ no anulable } (\beta \not\Rightarrow^* \lambda) \\ \text{Primeros}(\beta) \cup \text{Siguientes}(A) & \text{si } \beta \text{ anulable } (\beta \Rightarrow^* \lambda) \end{cases}$$

Algunos problemas y cómo resolverlos

- Ambigüedad por SDs no disjuntos

Algunos problemas y cómo resolverlos

- Ambigüedad por SDs no disjuntos
  - factorización a la izquierda (*left-factorization*)

Algunos problemas y cómo resolverlos

- Ambigüedad por SDs no disjuntos
  - factorización a la izquierda (*left-factorization*)
- Recursión a izquierda

Algunos problemas y cómo resolverlos

- Ambigüedad por SDs no disjuntos
  - factorización a la izquierda (*left-factorization*)
- Recursión a izquierda
  - eliminación de la recursión inmediata

Algunos problemas y cómo resolverlos

- Ambigüedad por SDs no disjuntos
  - factorización a la izquierda (*left-factorization*)
- Recursión a izquierda
  - eliminación de la recursión inmediata
  - eliminación de la recursión no inmediata

Factorización a la izquierda

Si tenemos producciones de la forma
$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ donde $\alpha \neq \lambda$, las reemplazamos por

$$A \to \alpha A' \mid \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$$
$$A' \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Eliminación de la recursión inmediata

Si tenemos producciones de la forma $A \to A\alpha \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_k$ las reemplazamos por

$$A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_k A'$$
$$A' \to \alpha A' \mid \lambda$$

Eliminación de la recursión no inmediata

1. Numerar los no terminales $A_1, \ldots, A_n$
2. Para $i \to 1$ a n
   1. Para $j \to 1$ a $i-1$
      1. Reemplazar $A_i \to A_j\gamma$ por por $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ (donde $A_j \to \delta_1 \mid \ldots \mid \delta_k \in P$)
   2. Fin Para
   3. Eliminar la recursión inmediata de A_i
3. Fin Para

Ejemplo

Dada la siguiente gramática para la declaración de tipos en un lenguaje de programación:
$G_1 = \langle \{S_0, T, S\}, \{\texttt{array}, \texttt{[}, \texttt{]}, \texttt{of}, \texttt{..}, \texttt{int}, \texttt{char}, \texttt{num}, \uparrow\}, S_0, P \rangle$, con P:

$$
\begin{array}{lcl}
S_0 & \longrightarrow & T \\
T & \longrightarrow & S \mid \uparrow S \mid \texttt{array of } T \mid \texttt{array [} S \texttt{] of } T \\
S & \longrightarrow & \texttt{int} \mid \texttt{char} \mid \texttt{num..num}
\end{array}
$$

- Hacer un parser descendente recursivo para esta gramática. Si no es posible, corregirla previamente
- La cadena $\texttt{array [num..num] of} \uparrow \texttt{char} \in L(G_1)$?
- ¿Cuál es la secuencia de derivaciones? ¿Cuál es el árbol sintáctico?

Ejemplo: Gramática corregida

- No es posible hacer un parser descendente recursivo predictivo para la gramática, porque hay solapamiento entre los SDs de dos reglas del no terminal T
- Se lo resuelve aplicando factorización a la izquierda, para crear G_1'

$G_1' = \langle \{S_0, T, T', S\}, \{\texttt{array}, \texttt{[}, \texttt{]}, \texttt{of}, \texttt{..}, \texttt{int}, \texttt{char}, \texttt{num}, \uparrow\}, S_0, P \rangle$, con P:

$$
\begin{array}{lcl}
S_0 & \longrightarrow & T \\
T & \longrightarrow & S \mid \uparrow S \mid \texttt{array}\, T' \\
T' & \longrightarrow & \texttt{of}\ T \mid \texttt{[}S\texttt{]}\ \texttt{of}\ T \\
S & \longrightarrow & \texttt{int} \mid \texttt{char} \mid \texttt{num..num}
\end{array}
$$

Ejemplo: Gramática corregida

$G_1' = \langle \{S_0, T, T'S\}, \{\texttt{array}, \texttt{[}, \texttt{]}, \texttt{of}, \texttt{..}, \texttt{int}, \texttt{char}, \texttt{num}, \uparrow\}, S_0, P \rangle$, con P:

$$
\begin{array}{lcl}
S_0 & \longrightarrow & T \\
T & \longrightarrow & S \mid \uparrow S \mid \texttt{array}\, T' \\
T' & \longrightarrow & \texttt{of}\ T \mid \texttt{[}S\texttt{]}\ \texttt{of}\ T \\
S & \longrightarrow & \texttt{int} \mid \texttt{char} \mid \texttt{num..num}
\end{array}
$$

Ejemplo: Gramática corregida y SDs

$G_1' = \langle \{S_0, T, T', S\}, \{\texttt{array}, \texttt{[}, \texttt{]}, \texttt{of}, \texttt{..}, \texttt{int}, \texttt{char}, \texttt{num}, \uparrow\}, S_0, P \rangle$

Regla			SD
S_0	$\longrightarrow$	T	$\{\texttt{int}, \texttt{char}, \texttt{num}, \uparrow, \texttt{array}\}$
T	$\longrightarrow$	S	$\{\texttt{int}, \texttt{char}, \texttt{num}\}$
T	$\longrightarrow$	$\uparrow S$	$\{\uparrow\}$
T	$\longrightarrow$	$\texttt{array}\, T'$	$\{\texttt{array}\}$
T'	$\longrightarrow$	$\texttt{of}\ T$	$\{\texttt{of}\}$
T'	$\longrightarrow$	$\texttt{[}S\texttt{]}\ \texttt{of}\ T$	$\{\texttt{[}\}$
S	$\longrightarrow$	$\texttt{int}$	$\{\texttt{int}\}$
S	$\longrightarrow$	$\texttt{char}$	$\{\texttt{char}\}$
S	$\longrightarrow$	$\texttt{num..num}$	$\{\texttt{num}\}$

Ejemplo: Parser descendente recursivo predictivo

```
Proc T()
    if tc in { int, char, num }
        S();
    else if tc in { ^ }
            match('^');
            S();
         else if tc in { array }
                 match('array');
                 T_p();
              else
                 error();
End

Proc T_p()
    if tc in { of }
        match('of');
        T();
    else if tc in { [ }
            match('['); S(); match(']');
            match('of');
            T();
         else
            error();
End
```

```
Proc S()
    if tc in { int }
       match('int');
    else if tc in { char }
            match('char');
         else if tc in { num }
                 match('num');
                 match('..');
                 match('num');
              else
                 error();
End

Proc Main();
    T();
    match('$');
    accept();
End
```

Teoría de Lenguajes

Intro compiladores + Analizadores Sintácticos Descendentes

Christian G. Cossio Mercado

DC-UBA

1er Cuatrimestre 2017