



Analizador sintáctico

Diseño de Compiladores

Dr. Víctor de la Cueva

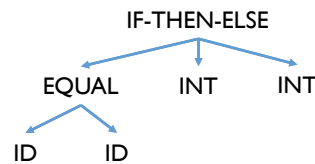
vcueva@itesm.mx

Parseo

- Es un proceso que tiene como objetivo realizar el **análisis sintáctico** o **gramatical** de un programa.
 - Debe asegurarse que el programa tiene una **estructura** que cumple con las **reglas sintácticas** del lenguaje
 - Recibe como entrada la secuencia de **tokens** y entrega como salida el **árbol sintáctico**
 - Tiene la responsabilidad de **detectar errores** en la sintaxis
- **Árbol sintáctico**: es una estructura para representar la estructura sintáctica del programa, también se le conoce como **árbol de análisis gramatical**.

Ejemplo

- Si una parte del programa es:
`if x = y then 1 else 2 end`
- El Lexer da como salida los siguientes tokens:
`IF ID EQUAL ID THEN INT ELSE INT END`
- El Parser recibe como entra estos token y da como salida el siguiente árbol:



Herramientas

- Se requieren al menos dos herramientas:
 - Una que nos ayude a especificar las reglas sintácticas del lenguaje
 - Otra que nos ayude a realizar la implementación de dichas reglas en un parser
 - Desde luego que puede ser la misma
- Los lenguajes regulares son unos de los lenguajes más simples (tienen muchas aplicaciones)
 - Muchos lenguajes, incluyendo los de programación, no son regulares: e.g. paréntesis balanceados: $\{()^i \mid i \geq 0\}$

Representación

- Los lenguajes de programación tienen una estructura recursiva (e.g. una expresión se define en función de una expresión)
- Las CFG son una notación natural para representar esta naturaleza recursiva.

Gramáticas libres de contexto (CFG)

- Una CFG consiste en:
 - Un conjunto de terminales T
 - Un conjunto de no-terminales N
 - Un símbolo de inicio S ($S \in N$)
 - Un conjunto de producciones $X \rightarrow Y_1 \dots Y_n$
 - $X \in N$
 - $Y_i \in N \cup T \cup \{\epsilon\}$

Reglas

- Las producciones pueden verse como **reglas**.
- El proceso completo es:
 1. Iniciar con el string con sólo el símbolo inicial **S**
 2. Reemplazar cada no-terminal **X** por el lado derecho de **alguna** producción $X \rightarrow Y_1 \dots Y_n$
 3. Repetir 2 hasta que no haya no-terminales
- Si al sustituir se tiene la secuencia de strings:
 - $\alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \dots \rightarrow \alpha_n$
 - Se dice que el string α_0 se reescribe como α_n en **n** pasos:
 $\alpha_0 \xrightarrow{*} \alpha_n \ (n \geq 0)$

Lenguaje

- Definición: Sea **G** una CFG con símbolo inicial **S**. Entonces, el lenguaje **L(G)** de G es:

$$\{a_1 \dots a_n \mid \forall i \ a_i \in T \wedge S \xrightarrow{*} a_1 \dots a_n\}$$
- En las aplicaciones de Lenguajes de Programación, los **terminales** de la CFG son los **tokens**.

Comentarios

- Se usa **EBNF** para representar la gramática del lenguaje.
- La idea de las CFG es excelente pero la membresía es “Sí” o “No” y en realidad se requiere como salida el *parse-tree* (este mecanismo es adicional).
- Necesita un buen manejador de errores, dando retroalimentación al programador.
- Se requiere implementar una CFG para crear el parser.
- La **forma** que tenga la **gramática** es muy importante:
 - Muchas gramáticas generan el mismo lenguaje
 - Las implementaciones son sensibles a la gramática

Derivaciones

- Es una secuencia de producciones
- Puede ser dibujada como un árbol (*parse tree*)
 - El símbolo inicial es la raíz
 - Para una producción $X \rightarrow Y_1 \dots Y_n$ agregar hijos $Y_1 \dots Y_n$ al nodo X . AA 3.5
- En el *parse tree*:
 - Las hojas son terminales
 - Los nodos interiores son no-terminales
 - El recorrido en inorden de las hojas nos da el string de entrada
 - El parse tree muestra la asociación de las operaciones, el string de entrada no (e.g. cuál se hace primero, * o +)
- ¿Varias derivaciones dan el mismo árbol?
 - rmd o lmd AA 4

Ambigüedad

- Se dice que una gramática es **ambigua** si existe más de un árbol de derivación para algún string. AA 4
 - La ambigüedad es **mala** para los lenguajes ya que el significado de las hojas para algunos programas queda mal definido AA 4.5
- Hay muchas forma de **eliminar la ambigüedad** para una gramática:
 - El método más directo es **reescribir** la gramática para quitarla
 - La nueva gramática debe generar el **mismo lenguaje** pero **sólo un parse tree** para cada string (el nuevo árbol no es exactamente igual pero general el mismo lenguaje)

AA 4.5, 5.5

Notas

- El problema de ver si una gramática es ambigua o no es **indecidable**.
 - Existen algunas herramientas que lo logran para ciertas gramáticas.
- **No existe una forma automática** de quitar la ambigüedad.
- Muchas implementaciones deciden usar la gramática ambigua:
 - Es más natural
 - Se deben colocar ciertas **reglas de precedencia y asociación** para guiar el árbol (declaraciones de desambigüedad)

AA 6 y 6.5

Errores

- Los compiladores tienen dos propósitos:
 - Traducir los programas válidos
 - Detectar los programas inválidos (y guiar al usuario sobre cómo hacerlos válidos)
- Existe una gran cantidad de errores posibles:

Tipo de error	Ejemplo	Detectado por
Léxico	... \$...	Lexer
Sintáctico	... x*% ...	Parser
Semántico	... int x; y = x[3]; ...	Typechecker
Correctez	Tu programa favorito	Tester/Usuario (un bug)

Manejo de errores

- Los requisitos de un buen manejador de errores:
 - Reportar los errores **exacta** y **claramente**
 - **Recuperarse** de un error rápidamente
 - **No bajar la velocidad** de compilación de un código válido
- Hay diferentes tipos de manejo de errores:
 - **Modo pánico**
 - Producciones de error
 - Corrección automática local o global

Modo pánico

- Es el modo más simple y el más popular.
- Cuando se detecta un error:
 - Desechar los token hasta encontrar **uno con un rol claro**
 - Continuar desde aquí
- Busca tokens de sincronización:
 - Típicamente los terminadores de estatutos o expresiones (e.g. `;;`, `end`, `endfor`)

AA 7.5

Árbol Sintáctico Abstracto (AST)

- Un parser traza la derivación de una secuencia de tokens.
- El resto de la compilación necesita una **representación estructural del programa**.
- El AST es como un *parser tree* pero **ignora algunos detalles**:
 - Cuando un nodo sólo tiene un sucesor se sustituye por él
 - Los paréntesis son muy importantes en el parser (muestran la asociación) pero una vez hecho el parseo no se requieren
 - El AST hace más reducciones

AA 8.5

Algoritmos de parseo

- Las CFG son una excelente herramienta para representar la gramática de los lenguajes de programación.
- Su implementación se basa en ellas y se clasifica en dos tipos de algoritmos:
 - Top-down: forma el AST de la raíz hacia abajo
 - Descendente recursivo
 - Predictivo (gramáticas $LL(k)$): entrada L -R, derivación L
 - Bottom-up
 - SLR (gramáticas $LR(k)$)

Top-Down Parsing

◦ **DESCENDENTE RECURSIVO**

Recursive Descent Parsing

- Es un algoritmo de parseo Top-Down
 - El parse tree es contruido:
 - Desde arriba (top)
 - De izquierda a derecha
 - Los terminales son revisados en el orden de aparición en el token stream
- Algoritmo:
 - Inicia con el no-terminal de nivel superior
 - Si la producción falla se hace un *backtracking* para probar producciones alternativas

AA 9

Implementación del algoritmo

- Sea TOKEN el tipo token
 - INT, OPEN, CLOSE, PLUS, TIMES, son instancias del tipo TOKEN
- Sea la variable global `next` un apuntador al siguiente token de entrada (\uparrow).
- Se definen algunas funciones booleanas que chequen un match de:
 - Un token terminal dado tok:


```
bool terminal(TOKEN tok) {return *next++ == tok}
```
 - La n-ésima producción de S:


```
bool Sn( ) {...}
```

 checa el éxito de una producción S
 - Intentar todas las producciones de S:


```
bool S( ) {...}
```

 tiene éxito si alguna producción de S tiene éxito

Ejemplo

- Implementación de la siguiente gramática:
 $E \rightarrow T \mid T + E$
 $T \rightarrow \text{int} \mid \text{int} * T \mid (E)$
- Para iniciar el parser:
 - Inicializa `next` al apuntador del primer token
 - Invoca `E()`
- Muy simple de implementar a mano.

AA 10

Limitaciones del algoritmo

- Pruebe la gramática implementada con `int * int`
- El problema es que no hay *backtracking* cuando la producción tiene éxito.
 - Eso significa que el algoritmo presentado **no es completamente general**
 - Sin embargo, es suficiente para gramáticas en las que para algún no terminal, **a lo más una producción** puede tener éxito
 - La gramática puede ser **reescrita** para trabajar con el algoritmo presentado:
 - Haciendo **factorización por la izquierda**

AA 10.5

Recursión por la izquierda

- Implemente la gramática $S \rightarrow Sa$
- $S()$ se va a un loop infinito.
- La razón es que la gramática es *left-recursive*
- Una gramática *left-recursive* tiene un no-terminal S de la forma $S \xrightarrow{+} S\alpha$ para alguna α .
- El algoritmo *Recursive Descent* no trabaja con este tipo de gramáticas.
 - Es un problema, pero no es muy grave

AA II

Eliminando la recursión por la izquierda

- En general, la gramática que genera todos los strings que inician con $\beta_1 | \dots | \beta_m$ y continúan con varias instancias de $\alpha_1, \dots, \alpha_n$, es decir:

$$S \rightarrow S\alpha_1 | \dots | S\alpha_n\beta_1 | \dots | \beta_m$$

- Se puede reescribir a una gramática con *right-recursion*:

$$S \rightarrow \beta_1 S' | \dots | \beta_m S'$$

$$S' \rightarrow \alpha_1 S' | \dots | \alpha_n S' | \epsilon$$

AA II
AA II.5

NOTA: Ver bibliografía para un algoritmo general que se puede implementar para hacerlo automáticamente.



Analizador Descendente Recursivo

EJEMPLO DE IMPLEMENTACIÓN



El método básico descendente recursivo

- La idea del análisis sintáctico descendente recursivo es muy simple:
 - Observamos la regla gramatical para un no terminal A como una definición para un procedimiento que reconocerá una A.
 - El lado derecho de la regla gramatical para A especifica la estructura del código para este procedimiento:
 - La secuencia de terminales y no terminales es una selección corresponde a concordancias de la entrada y llamadas a otros procedimientos.
 - Las selecciones corresponden a las alternativas (sentencias case o if) dentro del código.

Ejemplo: gramática de expresión en BNF

$\text{exp} \rightarrow \text{exp opsuma term} \mid \text{term}$

$\text{opsuma} \rightarrow + \mid -$

$\text{term} \rightarrow \text{term opmult factor} \mid \text{factor}$

$\text{opmult} \rightarrow *$

$\text{factor} \rightarrow (\text{exp}) \mid \text{número}$

L 144 factor

L 145 match

Repetición y selección

- Consideremos como segundo ejemplo la regla gramatical (simplificada) para un sentencia **if**:

$\text{sent-if} \rightarrow \text{if } (\text{exp}) \text{ sentencia}$

$\mid \text{if } (\text{exp}) \text{ sentencia } \text{else } \text{sentencia}$

- En este ejemplo podríamos no distinguir de inmediato cuál es la regla seleccionada ya que ambas comienzan con el token **if**.
- En su lugar, debemos aplazar la decisión acerca de reconocer la parte **else** opcional hasta que veamos el token **else** en la entrada,
- De esta forma, el código correspondería más a la EBNF:

$\text{sent-if} \rightarrow \text{if } (\text{exp}) \text{ sentencia } [\text{else } \text{sentencia}]$

L 145

La notación EBNF

- La notación **EBNF** está diseñada para **reflejar** muy de cerca el **código real** de un analizador sintáctico descendente recursivo
- Una gramática deberá **siempre traducirse a EBNF** si se está utilizando este modo

NOTA: Aún cuando la gramática anterior es **ambigua** es natural escribir un analizador sintáctico que haga concordar cada token **else** tan pronto como se encuentre en la entrada. Esto corresponde precisamente a la **regla de eliminación de la ambigüedad mas cercanamente anidada**.

Caso de una exp

- En BNF:

$$\text{exp} \rightarrow \text{exp opsuma term} \mid \text{term}$$
- Se observa de inmediato que es recursiva por la izquierda y llevaría a un ciclo infinito (se puede evitar pero hacerlo sería muy problemático)
- La solución es utilizar EBNF:

$$\text{exp} \rightarrow \text{term} \{ \text{opsuma term} \}$$
 - Las llaves expresan la repetición que se puede traducir al código con un ciclo (o bucle) L 146 exp
- De la misma forma se hace para term:

$$\text{term} \rightarrow \text{factor} \{ \text{opmult factor} \}$$
L 146 term

Pseudocódigo a código real

- Es muy importante tomar en cuenta que el pseudocódigo presentado se debe traducir a código real, para lo cual, muchas veces se requiere que los procedimientos regresen algo, lo que los convierte en **funciones**.
 - Si se desean hacer operaciones (e.g. implementar una **calculadora**) podría regresar un **entero** correspondiente a la evaluación
 - Si se está haciendo un **parser** debe regresar el AST

Construyendo el AST

- En realidad, la implementación de un parser simplemente responde a la pregunta de si un programa está bien escrito con un “Sí” o “No”.
- Si se desea que haga más cosas se le deben agregar en el código de los procedimientos.
- Para el caso de un AST, se le debe agregar la creación del árbol en cada nodo, por ejemplo:
 - **exp**: crear un nodo con **+** o **–** en la raíz y formar sus hijos
 - **ifStatement**: crear un nodo (de tipo **if**) con tres hijos:
 - **Condición** (nodo **exp**), **then** (nodo **sentencia**) y **else**, si es que existe (nodo **sentencia**)

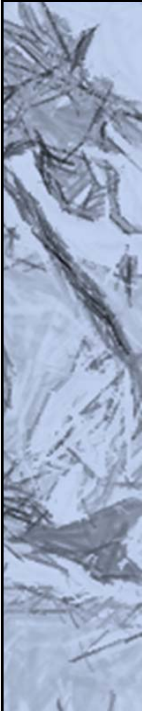
L 150 exp
L 151 if



Top-Down Parsing

PREDICTIVO

AA I2



Predictive parsing

- Es como el descendente-recursivo pero puede predecir cuál producción usar:
 - Viendo los siguientes (unos cuantos) tokens (*lookahead*)
 - Sin *backtracking*
 - También usa formas restrictivas de las gramáticas
 - Acepta lo que llamamos gramáticas LL(k)
 - L Left to right scan
 - L Left-most derivation
 - k k tokens of lookahead, (en la práctica $k=1$)

Predictivo vs descendente recursivo

- En descendente-recursivo:
 - En cada paso se usan muchas opciones de producción
 - Se usa *backtracking* para deshacer malas selecciones
 - Usa llamadas recursivas a las funciones
- En LL(1):
 - En cada paso sólo se tiene **una opción** de producción
 - Requiere una gramática *left-factor*, en BNF, con la idea de eliminar prefijos comunes de múltiples producciones
 - Esta gramática se utiliza para construir una **tabla de parseo**
 - Todas las entradas que no tienen producción son errores
 - En lugar de llamadas recursivas usa un *stack*

Tabla de análisis sintáctico

- La tabla contiene producciones y está indizada en las filas por No-terminales y en las columnas por tokens, incluyendo el \$.
- Agregamos a la tabla **M** opciones de producción de acuerdo con las siguientes reglas:
 - Si $A \rightarrow \alpha$ es una opción de producción, y existe una derivación $\alpha \xRightarrow{*} a\beta$, donde a es un token, entonces se agrega $A \rightarrow \alpha$ a la entrada $M[A, a]$.
 - Si $A \rightarrow \alpha$ es una opción de producción, y existe una derivaciones $\alpha \xRightarrow{*} \varepsilon$ y $S \xRightarrow{*} \beta A a \gamma$, donde S es el símbolo inicial y a es un token (o \$), entonces se agrega $A \rightarrow \alpha$ a la entrada $M[A, a]$.

Explicaciones de las reglas

- En la regla 1, dado un token a en la entrada, deseamos seleccionar un regla $A \rightarrow \alpha$ si α puede producir una a para comparar.
- En la regla 2, si A deriva la cadena vacía (vía $A \rightarrow \alpha$), y si a es token que puede venir legalmente después de A en una derivación, entonces deseamos seleccionar $A \rightarrow \alpha$ para hacer que A desaparezca. Un caso especial de la regla 2 ocurre cuando $\alpha = \epsilon$.

NOTA: Estas reglas son difíciles de implementar de manera directa pero existe un algoritmo que lo hace automáticamente usando los conjuntos FIRST y FOLLOW.

Ejemplos

$$S \rightarrow (S) S \mid \epsilon$$

L 152

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

AA 12.5

Algoritmo de análisis LL(1) basado en tabla

```

/* Suponer que $ marca la parte inferior de la pila y el final de la entrada */
while el top ≠ $ and el siguiente token ≠ $ do
  if el top = a and siguiente token = a
    then /* concuerda */
      pop a la pila;
      avanzar la entrada;
    else if top = A and siguiente token = a and  $M[A,a] = A \rightarrow X_1 X_2 \dots X_n$ 
      then /* generar */
        pop de la pila;
        push de  $X_1 X_2 \dots X_n$  a la pila;
      else error;
  if top = $ and siguiente token = $
    then aceptar
  else error;

```

Otro algoritmo Parseo Predictivo

```

Inicializar el stack = <S$> y next
repeat
  case stack of:
    <X, rest> : if  $M[X, *next] = Y_1 \dots Y_n$ 
      then stack ← < $Y_1 \dots Y_n$  rest>;
      else error();
    <t, rest> : if t == *next++
      then stack ← <rest>;
      else error();
until stack == <>

```

Conjuntos FIRST y FOLLOW

- ¿Cómo construir fácilmente la tabla para LL(1)?
- Considere el no-terminal A , la producción $A \rightarrow \alpha$ y el token t : hacemos $M[A, t] = \alpha$ en dos casos:
 - Si $\alpha \rightarrow^* t\beta$
 - α puede derivar a t en la primera posición
 - Decimos que $t \in \text{FIRST}(\alpha)$
 - Si $A \rightarrow \alpha$, $\alpha \rightarrow^* \varepsilon$ y $S \rightarrow^* \beta A t \delta$
 - Es útil si el stack tiene a A , la entrada es t y A no puede derivar a t
 - En este caso, sólo hay una opción para deshacerse de A (derivando ε)
 - Sólo puede ser si t puede seguir a A en al menos una derivación
 - Decimos que $t \in \text{FOLLOW}(A)$

FIRST

- Definición: $\text{FIRST}(X) = \{t \mid X \rightarrow^* t\alpha\} \cup \{\varepsilon \mid X \rightarrow^* \varepsilon\}$
- Algoritmo:
 1. $\text{FIRST}(t) = \{t\}$ t es un terminal
 2. $\varepsilon \in \text{FIRST}(X)$ X es un no-terminal
 - Si $X \rightarrow \varepsilon$
 - Si $X \rightarrow A_1 \dots A_n$ y $\varepsilon \in \text{FIRST}(A_i)$ para toda $1 \leq i \leq n$
 3. $\text{FIRST}(\alpha) \subseteq \text{FIRST}(X)$ si
 - $X \rightarrow A_1 \dots A_n \alpha, \gamma$
 - $\varepsilon \in \text{FIRST}(A_i)$ para toda $1 \leq i \leq n$

AA 13.5

FOLLOW

- Definición: $\text{FOLLOW}(X) = \{ t \mid S \rightarrow^* \beta X t \delta \}$
- Intuición:
 - Si $X \rightarrow AB$ entonces $\text{FIRST}(B) \subseteq \text{FOLLOW}(A)$ y $\text{FOLLOW}(X) \subseteq \text{FOLLOW}(B)$
 - Si $B \rightarrow^* \varepsilon$ entonces $\text{FOLLOW}(X) \subseteq \text{FOLLOW}(A)$
 - Si S es el símbolo inicial entonces $\$ \in \text{FOLLOW}(S)$
- Algoritmo:
 1. $\$ \in \text{FOLLOW}(S)$
 2. $\text{FIRST}(\beta) - \{\varepsilon\} \subseteq \text{FOLLOW}(X)$
 - Para cada producción $A \rightarrow \alpha X \beta$
 3. $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(X)$
 - Para cada producción $A \rightarrow \alpha X \beta$ donde $\varepsilon \in \text{FIRST}(\beta)$

AA 14

Tabla de parseo a partir de FIRST y FOLLOW

- Objetivo: Construir una tabla M para una CFG G .
- Para cada producción $A \rightarrow \alpha$ en G hacer:
 - Para cada terminal $t \in \text{FIRST}(\alpha)$ hacer:
 - $M[A, t] = \alpha$
 - Si $\varepsilon \in \text{FIRST}(\alpha)$, para cada $t \in \text{FOLLOW}(A)$ hacer:
 - $M[A, t] = \alpha$
 - Si $\varepsilon \in \text{FIRST}(\alpha)$ y $\$ \in \text{FOLLOW}(A)$ hacer:
 - $M[A, \$] = \alpha$

AA 14.5



Bottom-up

◦ **SIMPLE LR (SLR)**



Referencias

- A.V.Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd Pearson (2012).
- K.C. Loudon. *Contrucción de Compiladores: principios y práctica*. Thomson (2004).
- Alex Aiken. Compilers. Stanford Online (2018).
 - <https://lagunita.stanford.edu/courses/Engineering/Compilers/Fall2014/about>