Compiladores
Facultad de Ciencias UNAM
Análisis Sintáctico: *Parser top-down* [*]

Lourdes Del Carmen González Huesca [**]

24 de marzo de 2024

En el proceso de parsing, usualmente se puede realizar un retroceso o backtracking para encontrar la producción correcta en la construcción parcial del árbol. Alternativamente se puede diseñar un analizador **predictivo** que selecciona o predice la producción adecuada para construir un subárbol al procesar un número fijo de símbolos o tokens al mismo tiempo.

El parsing recursivo descendente es un método top-down en el cual un conjunto de procedimientos recursivos es usado para procesar el programa fuente o la secuencia de tokens obtenida en el análisis léxico.

Analizador LL

Las clases de gramáticas para las que se puede construir un analizador predictivo son llamadas gramáticas LL. En el tipo de parsers LL se puede predecir la regla que construye de forma única un subárbol al seleccionar una regla que reescribe un símbolo no-terminal de la gramática (no ambigua). Esta clase de parsers realizan un análisis para reconocer una cadena al encontrar una derivación más a la izquierda y construir el árbol desde la raíz y hacia las hojas, creando los nodos en preorden.

El nombre **LL** corresponde a un análisis de izquierda a derecha (Left-to-right) y que construye una derivación por la izquierda (Leftmost).

El parsing predictivo analiza un número fijo k de símbolos subsecuentes por adelantado al símbolo o token actual. Dependiendo del número k se pueden clasificar las gramáticas en clases de parsers denotados por **LL(k)**. Analizaremos la clase (decidable) **LL(1)**, que sólo usa un símbolo o el token actual.

Definición 1 (Gramática **LL(1)**)**.** Decimos que una gramática pertenece a la clase **LL(1)** si para cualquier símbolo terminal a la izquierda de las producciones con más de dos reglas, los conjuntos de sus derivaciones son disjuntas, es decir para cada producción $A \to \alpha \mid \beta$ con $\alpha \neq \beta$:

1. $\textsc{First}(\alpha) \cap \textsc{First}(\beta) = \varnothing$
2. Si $\beta \to^* \varepsilon$ entonces $\textsc{First}(\alpha) \cap \textsc{Follow}(A) = \varnothing$
3. Análogamente si $\alpha \to^* \varepsilon$.

Ninguna gramática recursiva por la izquierda (con producciones de tipo $E \to Ew$) o ambigua puede ser **LL(1)**. Para algunas gramáticas que no pertenecen a esta clase es posible encontrar una gramática equivalente que sí lo sea usando transformaciones como factorización, eliminar producciones épsilon y

[*] Los ejemplos están tomados del libro "Compilers, Principles, Techniques and Tools" [1]

[**] Material revisado por el servicio social de Apoyo a la Docencia y Asesoría Académica en la Facultad de Ciencias de la UNAM con clave 2023-12/12-292, de Diana Laura Nicolás Pavia.

recursión. Sin embargo, existen grámaticas que no pueden transformarse a una **LL(1)** como es el caso de la gramática *dangling-else*.

Un analizador predictivo se basa en la idea de elegir la producción $A \rightarrow \alpha$ si el siguiente token a pertenece a $\textsc{First}(\alpha)$, por lo que reune toda la información necesaria en una tabla auxiliar o tabla de análisis o de parsing.

Tabla de análisis

Una tabla de análisis predictivo $M[X, a]$, donde X es un símbolo no terminal y a es un símbolo terminal, indica qué producción debe usarse si se quiere derivar una cadena $a\omega$ a partir de X. El procedimiento para construir dicha tabla se muestra a continuación considerando que existen producciones epsilon en la gramática:

Para cada producción $X \rightarrow \alpha$:

1. Para cada terminal a en $\textsc{First}(\alpha)$ agregar la entrada $M[X, a] = X \rightarrow \alpha$
2. Si $\varepsilon \in \textsc{First}(\alpha)$ entonces:
   Para cada $b \in \textsc{Follow}(X)$ agregar $M[X, b] = X \rightarrow \alpha$
   Si $\varepsilon \in \textsc{First}(\alpha)$ y # está en $\textsc{Follow}X$ entonces:
   Agregar $M[X, \#] = X \rightarrow \alpha$

Las entradas vacías de la tabla, indican que no existe una producción para derivar la entrada. Se puede construir la tabla de análisis para cualquier gramática, si es **LL(1)** cada entrada de la tabla tiene una sola producción o está vacía; si la gramática es recursiva o ambigua, tendrá entradas con múltiples producciones.

Algoritmo para LL

Para reconocer una cadena de una gramática **LL(1)** manejamos una pila cuyo estado inicial contiene S y el fondo de la pila es el símbolo de fin de cadena. El algoritmo construye una derivación izquierda y/o un árbol de sintaxis como objeto final. En cada paso se puede observar la derivación o la parte del árbol con w la (sub)cadena que se ha reconocido hasta el momento, la pila contiene una secuencia de símbolos α tal que $S \rightarrow^* w\alpha$. En cada iteración se considera el tope de la pila y el siguiente token, si se tiene un símbolo no terminal entonces se consulta la tabla para decidir qué producción utilizar, en otro caso se comparan los símbolos (*match*):

- Se inicia con a el primer símbolo de ω y $S\#$ el tope de la pila.
- Mientras la pila tenga elementos:
  sea X el tope de la pila, a el token actual y M la tabla de análisis.
  - Si X es un símbolo terminal y $X == a$ entonces se saca a X del tope de la pila y se obtiene el siguiente token
  - Si X es un símbolo terminal y $X! = a$ entonces hay un error
  - Si X es un símbolo no-terminal entonces:
    - $M[X, a]$ contiene una producción $X \rightarrow Y_1..Y_k$
      se saca a X del tope de la pila y se agrega a $Y_k...Y_1$ donde Y_1 es el nuevo tope;
      se construye el nodo en el árbol de derivación o se registra la producción como parte de la derivación
    - $M[X, a]$ no contiene una producción entonces hay un error

A continuación el pseudoalgoritmo:

```
let a be the first symbol of w ;
let X be the top stack symbol;
while ( X != # )  /* stack is not empty */
     {
      if ( X = a )
      then pop the stack and let a be the next symbol of w;
      else if ( X is a terminal ) error();
      else if ( M [X; a] is an error entry ) error();
      else if ( M [X; a] = X -> Y1Y2...Yk );
      {
              output the production X -> Y1Y2...Yk;
              pop the stack;
              push Yk, Yk-1, ..., Y1 onto the stack; /* with Y1 on top */
      }
      let X be the top stack symbol;
     }
```

Ejemplo 1 (Lenguaje de expresiones aritméticas sencillas)**.**

- Construcción de la tabla de parsing predictivo

$$
\begin{array}{llll}
E & \to & TE' & \\
E' & \to & +TE' & E' \to \varepsilon \\
T & \to & FT' & T \to F \\
T' & \to & *FT' & T' \to \varepsilon \\
F & \to & (E) & F \to id
\end{array}
$$

	E	E'	T	T'	F
First	$\{(, id\}$	$\{+, \varepsilon\}$	$\{(, id\}$	$\{*, \varepsilon\}$	$\{(, id\}$
Follow	$\{\#,)\}$	$\{\#,)\}$	$\{+, \#,)\}$	$\{+, \#,)\}$	$\{*, +, \#,)\}$

Non-terminal	Input Symbol					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ id			$F \to (E)$		

- Ejecución del algoritmo para input: $id + id * id$

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\, T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+\, \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+\, \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+\, TE'\$$	$+\, \mathbf{id} * \mathbf{id}\$$	output $E' \to +\, TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id} +$	$\mathbf{id}\, T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \to * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}\, T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Derivación por la izquierda (resultado del almacenamiento de las reglas en la pila) y que también puede generar un parse-tree que se construye por los nodos en preorden:

$$E \to TE' \to FT'E' \to idT'E' \to id\varepsilon E' \to id + TE' \to id + FT'E' \to id + idT'E' \to$$
$$id + id * FT'E' \to id + id * idT'E' \to id + id + *id\varepsilon \to id + id + *id\varepsilon\varepsilon = id + id * id$$

Parsers LL(k)

La clase de parsers fuertes **LL**(k) (**SLL**(k)) incluye aquellos analizadores donde los símbolos leídos por adelantado son suficientes para seleccionar la producción correcta en la gramática para construir el parse tree. Este tipo de analizadores generaliza el proceso anterior al considerar k símbolos por adelantado. Crea y usa una tabla de predicción de la misma forma salvo que esta tabla tiene mayor tamaño al considerar las columnas como cadenas y no sólo un símbolo.

Veamos las generalizaciones de la teoría de cadenas para este proceso así como las generalizaciones de las funciones FIRST y FOLLOW:

Dado un alfabeto Σ y una palabra $\omega = a_1 a_2 \dots a_n$ con $a_i \in \Sigma$ se define

- k-prefijo de ω como
$$\omega|_k = \begin{cases} a_1 a_2 \dots a_n & \text{si} \quad n \leqslant k \\ a_1 a_2 \dots a_k & \text{en otro caso} \end{cases}$$
- k-concatenación $\odot_k : \Sigma^\star \times \Sigma^\star \to \Sigma^{\leqslant k}$, con $\Sigma^{\leqslant k} = \bigcup_{i=0}^{k} \Sigma^i$, como
$$a \odot_k b = (ab)|_k$$

Definición 2 (Función FIRST_k)**.** Función que calcula el conjunto de prefijos de tamaño k que se pueden derivar de α:

$$\text{FIRST}_k(\alpha) = \{\omega|_k \mid \alpha \to^\star \omega\}$$

$$\text{FIRST}_k(A) = \bigcup \{\text{FIRST}_k(A_1) \odot_k \cdots \odot_k \text{FIRST}_k(A_n) \mid A \to A_1 A_2 \dots A_n,\ A_i\, var\}$$

Definición 3 (Función $\textsc{Follow}_k$)**.** La función $\textsc{Follow}_k$ para un símbolo no-terminal X calcula el conjunto de símbolos terminales de a lo más tamaño k que pueden seguir directamente a la variable X:

$$\textsc{Follow}_k(X) = \{\omega \in \Sigma^\star \mid S \to^\star vX\gamma, \text{ y } \omega \in \textsc{First}_k(\gamma\#)\}$$

Definición 4. Decimos que una gramática pertenece a la clase **SLL(k)** si para cualquier símbolo no-terminal con más de dos reglas de producción $A \to \beta \mid \gamma$ con $\beta \neq \gamma$ siempre sucede que

$$\textsc{First}_k(\beta) \odot_k \textsc{Follow}_k(A) \cap \textsc{First}_k(\gamma) \odot_k \textsc{Follow}_k(A) = \varnothing$$

Toda gramática **LL(1)** es una gramática **SLL(1)**. Pero para $k > 1$, una gramática **LL(k)** no necesariamente es también una **SLL(k)**, dado que el conjunto $\textsc{Follow}_k(A)$ contiene todas las palabras que se pueden generar desde A hacia la izquierda.

La tabla de predicción M tiene por renglones las variables de la gramática y como columnas cadenas de terminales de longitud k, almacena producciones en las entradas $M[X, \omega]$ que se determinan de la siguiente forma:

> Sean $Y \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_r$ las producciones de la gramática correspondientes a la variable Y.
> Dado que los conjuntos $\textsc{First}_k(\alpha_i) \odot_k \textsc{Follow}_k(Y)$ son disjuntos en este tipo de gramáticas entonces
>
> - para cada $\omega \in \textsc{First}_k(\alpha_1) \odot_k \textsc{Follow}_k(Y) \cup \textsc{First}_k(\alpha_2) \odot_k \textsc{Follow}_k(Y) \cup \cdots \cup \textsc{First}_k(\alpha_r) \odot_k \textsc{Follow}_k(Y)$ se tiene que
>
> $$M[X, \omega] = \alpha_i \quad \text{ si y sólo si } \quad \omega \in \textsc{First}_k(\alpha_i) \odot_k \textsc{Follow}_k(Y)$$
>
> - en otro caso se deja la entrada de la tabla vacía o se incluye una rutina para manejar el error

La construcción de parsers **LL(k)** se puede restringir a gramáticas **SLL(k)**. La coincidencia de los símbolos leidos por adelantado y las columnas en la tabla de parsing hacen que la regla de producción sea única. Pero esto hace que la tabla pueda ser muy grande para $k > 1$, por lo que los parsers k-predictivos son evitados en la práctica y se usan más los de tipo **LL(1)** que son **SLL(1)**.

Manejo de errores

Durante el análisis sintáctico pueden surgir errores como un símbolo faltante o algunos paréntesis no balanceados. El objetivo es detectar el error y reportarlo de forma clara; una vez hecho esto, el analizador debe 'recuperarse' y resumir o restaurar el análisis tan rápido como sea posible sin agregar complejidad al proceso. A continuación se describen dos estrategias.

Modo pánico Esta estrategia se caracteriza por ignorar símbolos ya sea de la cadena de entrada o de la pila.

Si dado el símbolo no-terminal X, la entrada $M[X, a]$ está vacía, el modo pánico puede descartar tokens de la cadena de entrada hasta que uno pertenezca al conjunto de sincronización $SYNCH(X)$ que permite restaurar el estado del parser. La decisión de los elementos que pertenecen a $SYNCH(X)$ puede guiarse por las siguientes estrategias:

1. Incluir elementos en $\textsc{Follow}(X)$. Si encontramos un token que pertenzca a este conjunto, podemos saltarnos la derivación de X y resumir el análisis. El error a reportar involucra la derivación de X: "Se espera un ..."

2. En el caso de lenguajes que definen una jerarquía, se pueden incluir símbolos de la jerarquía superior.

3. Incluir elementos en $\textsc{First}(X)$. Si encontramos un token que pertenezca a este conjunto, podemos resumir el análisis en ese estado. Se reporta el error con los tokens que fueron descartados: "Tokens inesperados ..."

4. Si X puede derivar la cadena vacía, usar esa producción.

Cuando el tope de la pila es un símbolo terminal que no coincide con el siguiente token, la estrategia más simple es modificar la pila al sacar el topo y marcar un error de inserción: "Se esperaba token ..."

Por ejemplo, para la gramática de expresiones aritméticas, la cadena $n + *n$ es errónea:

reconocido	*stack*	*entrada*	
ε	$E\#$	$n + *n\#$	
ε	$TE'\#$	$n + *n\#$	
ε	$FT'E'\#$	$n + *n\#$	
ε	$nT'E'\#$	$n + *n\#$	
n	$T'E'\#$	$+ * n\#$	
n	$\varepsilon E'\#$	$\varepsilon + *n\#$	
n	$E'\#$	$+ * n\#$	
n	$+TE'\#$	$+ * n\#$	
$n+$	$TE'\#$	$*n\#$	error
			$*$ no pertenece ni a $\textsc{Follow}$ ni a $\textsc{First}$
			descartar el token con mensaje
			error : "Simbolo * inesperado"
$n+$	$TE'\#$	$n\#$	n está en el $\textsc{First}$(T), resumir/restaurar parser
$n+$	$FT'E'\#$	$n\#$	
$\vdots$	$\vdots$	$\vdots$	
$n+$	$n\#$	$n\#$	
$n + n$	$\#$	$\#$	pila vacía, termina el análisis

Recuperación a nivel de frase En esta estrategia las entradas vacías de la tabla de análisis son llenadas con funciones que manipulen (usualmente agreagando símbolos) la pila y/o la entrada para corregir el estado actual y resumir o restaurar el análisis. En el ejemplo anterior, la entrada de la tabla $M[T, *]$ puede tener una función que inserte un token de identificador n con el objetivo de reconocer la expresión completa:

reconocido	*stack*	*entrada*	
$\vdots$	$\vdots$	$\vdots$	
$n+$	$TE'\#$	$*n\#$	*error* : T esperaba un identificador
			$M[T, *]$ crea un identificador nuevo **n**
			se reporta el identificador
$n+$	$TE'\#$	$\mathbf{n} * n\#$	resume análisis
$n+$	$FT'E'\#$	$\mathbf{n} * n\#$	
$n+$	$nT'E'\#$	$\mathbf{n} * n\#$	
$n + \mathbf{n}$	$T'E'\#$	$*n\#$	
$n + \mathbf{n}$	$*FT'E'\#$	$*n\#$	
$n + \mathbf{n}*$	$FT'E'\#$	$n\#$	
$n + \mathbf{n}*$	$nT'E'\#$	$n\#$	
$n + \mathbf{n} * n$	$T'E'\#$	$\#$	
$n + \mathbf{n} * n$	$E'\#$	$\#$	
$n + \mathbf{n} * n$	$\#$	$\#$	

Cabe resaltar que esta estrategia deriva una cadena distinta al corregirla, sin embargo, las funciones a usar deben ser diseñadas con cuidado para evitar caer en un loop infinito cuando no se descarta un elemento de la pila o entrada. La implementación de las funciones es decisión del diseñador(a) del compilador y está fuertemente relacionada a la gramática.

Referencias

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Pearson Education Inc., Second edition, 2007.

[2] Jean-Christophe Filliâtre. Curso Compilation (inf564) école Polytechnique, Palaiseau, Francia. `http://www.enseignement.polytechnique.fr/informatique/INF564/`, 2018. Material en francés.

[3] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Ellis Horwood, USA, 1990.

[4] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. Springer Publishing Company, Incorporated, 2nd edition, 2012.

[5] Frank Pfenning. Notas del curso (15-411) Compiler Design. `https://www.cs.cmu.edu/~fp/courses/15411-f14/`, 2014.

[6] François Pottier. Presentaciones del curso Compilation (inf564) École Polytechnique, Palaiseau, Francia. `http://gallium.inria.fr/~fpottier/X/INF564/`, 2016. Material en francés.

[7] Michael Lee Scott. *Programming Language Pragmatics*. Morgan-Kauffman Publishers, Third edition, 2009.

[8] Yunlin Su and Song Y. Yan. *Principles of Compilers, A New Approach to Compilers Including the Algebraic Method*. Springer-Verlag, Berlin Heidelberg, 2011.

[9] Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. *Compiler Design*. Springer-Verlag Berlin Heidelberg, 2013.

[10] Steve Zdancewic. Notas del curso (CIS 341) - Compilers, Universidad de Pennsylvania, Estados Unidos. `https://www.cis.upenn.edu/~cis341/current/`, 2018.