Compiladores 24-2
Análisis Sintáctico: parsers top-down

Lourdes del Carmen González Huesca
luglzhuesca@ciencias.unam.mx

Facultad de Ciencias, UNAM

28 febrero 2024

Análisis sintáctico

Top-down parsing

- Top-down parsing para gramáticas tipo **LL** (*scan input from left to right & left-most derivation*).
- Un parser **LL(k)** es uno también llamado predictivo, en donde se revisan *k* tokens por adelantado

  para una variable o un símbolo no-terminal, el símbolo leído por adelantado determina de forma única la producción a aplicar

Análisis sintáctico

Top-down parsing

- Top-down parsing para gramáticas tipo **LL** (*scan input from left to right & left-most derivation*).
- Un parser **LL(k)** es uno también llamado predictivo, en donde se revisan k tokens por adelantado
  para una variable o un símbolo no-terminal, el símbolo leído por adelantado determina de forma única la producción a aplicar
- Analizaremos la clase (decidible) **LL(1)**, que solo considera un símbolo por adelantado (el siguiente token).
- Top-down parsing se puede entender como la forma de construir un parse-tree iniciando desde la raíz y creando los nodos en preorden.
- **Observación:** Ninguna gramática recursiva por la izquierda (con producciones de tipo $E \to Ew$) o ambigua puede ser **LL(1)**.

Análisis sintáctico

Definición (Gramática **LL(1)**)

Decimos que una gramática pertenece a la clase **LL(1)** si para cualquier símbolo no-terminal con más de dos reglas, los conjuntos de sus derivaciones son disjuntas, es decir para cada variable con producciones $A \to \alpha \mid \beta$ y $\alpha \neq \beta$:

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

Recordemos que

$$\textsc{First}(\alpha) = \{a \in \Sigma \mid \exists w, \alpha \to^* aw\}$$

$$\textsc{Follow}(X) = \{a \in \Sigma \mid A \to vXaw,\ v, w \in \Gamma \cup \Sigma\}$$

Una gramática es de tipo **LL** si para todo símbolo no-terminal A

si $S \rightarrow^{\star} xA\beta$ y $A \rightarrow \alpha_1$ y $A \rightarrow \alpha_2$ producciones distintas,

sucede que $\textsc{First}(\alpha_1\beta) \cap \textsc{First}(\alpha_2\beta) = \varnothing$

Análisis sintáctico

Top-down parsing predictivo

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

Análisis sintáctico

Top-down parsing predictivo

- Un analizador predictivo se basa en la idea de elegir la producción $A \to \alpha$ si el siguiente token t pertenece a $\textsc{First}(\alpha)$,
  por lo que reune toda la información necesaria en una tabla auxiliar.
- Una tabla de análisis predictivo definida como $M[X, a]$,
  donde X es un símbolo no-terminal y a es un símbolo terminal incluso $\#$,
  indica qué producción debe usarse si se quiere derivar una cadena $a\omega$ a partir de X.

Análisis sintáctico

Top-down parsing predictivo

- Un analizador predictivo se basa en la idea de elegir la producción $A \to \alpha$ si el siguiente token t pertenece a $\textsc{First}(\alpha)$,
  por lo que reune toda la información necesaria en una tabla auxiliar.
- Una tabla de análisis predictivo definida como $M[X, a]$,
  donde X es un símbolo no-terminal y a es un símbolo terminal incluso $\#$,
  indica qué producción debe usarse si se quiere derivar una cadena $a\omega$ a partir de X.
- El algoritmo selecciona una producción $A \to \alpha$ si el siguiente símbolo en la cadena de entrada está en $\textsc{First}(\alpha)$.
- En el caso de tener $\alpha = \varepsilon$ entonces se toma la misma producción.

Análisis sintáctico

Construcción de la tabla de parsing predictivo

INPUT: una gramática

la gramática original sin la regla $S \rightarrow E\#$ pero considerando a $\#$ como símbolo terminal

OUTPUT: una tabla de parsing M

cada entrada contiene las reglas que coinciden con la construcción de un subárbol del parse-tree

Análisis sintáctico

Construcción de la tabla de parsing predictivo

INPUT: una gramática

la gramática original sin la regla $S \to E\#$ pero considerando a $\#$ como símbolo terminal

OUTPUT: una tabla de parsing M

cada entrada contiene las reglas que coinciden con la construcción de un subárbol del parse-tree

Para cada producción $X \to \alpha$:

1. Para cada terminal a en $\text{FIRST}(\alpha)$ agregar la entrada $M[X, a] = X \to \alpha$
2. Si $\varepsilon \in \text{FIRST}(\alpha)$ entonces:
   Para cada $b \in \text{FOLLOW}(X)$ agregar $M[X, b] = X \to \alpha$
   Si $\varepsilon \in \text{FIRST}(\alpha)$ y $\#$ está en $\text{FOLLOW}(X)$ entonces:
   agregar $X \to \alpha$ a $M[X, \#]$
3. Si al realizar los pasos anteriores no se asigna producción alguna a $M[A, a]$ entonces asignar `error` o dejarla vacía.

Análisis sintáctico

Construcción de la tabla de parsing predictivo

- 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.
- Para una gramática **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.

Análisis sintáctico

Ejemplo: Construcción de la tabla de parsing predictivo

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

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

Análisis sintáctico

Ejemplo: Construcción de la tabla de parsing predictivo

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

	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	$F \to \mathrm{id}$			$F \to (E)$		

Análisis sintáctico

Algoritmo para LL

- Para reconocer una cadena de una gramática **LL(1)** mediante una máquina, manejamos una pila cuyo estado incial contiene al símbolo inicial de la gramática y el fondo de la pila es el símbolo de fin de cadena.

Análisis sintáctico

Algoritmo para LL

- Para reconocer una cadena de una gramática **LL(1)** mediante una máquina, manejamos una pila cuyo estado incial contiene al símbolo inicial de la gramática 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, si w es la cadena que se ha reconocido hasta el momento, la pila contiene una secuencia de símbolos α tal que $S \rightarrow^\star w\alpha$.

Análisis sintáctico

Algoritmo para LL

- Para reconocer una cadena de una gramática **LL(1)** mediante una máquina, manejamos una pila cuyo estado incial contiene al símbolo inicial de la gramática 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, si w es la cadena que se ha reconocido hasta el momento, la pila contiene una secuencia de símbolos α tal que $S \rightarrow^\star w\alpha$.
- 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.

Análisis sintáctico

Algoritmo para LL

INPUT: una cadena de entrada ω (secuencia de tokens) y
una tabla de parsing M para la gramática determinada G (def. lenguaje)

OUTPUT: Si $\omega \in \mathcal{L}(G)$ entonces devuelve una derivación por la izquierda de ω en otro caso devuelve un error.
También se puede obtener el *parse tree*.

Análisis sintáctico

Algoritmo para LL

INPUT: una cadena de entrada ω (secuencia de tokens) y
una tabla de parsing M para la gramática determinada G (def. lenguaje)

OUTPUT: Si $\omega \in \mathcal{L}(G)$ entonces devuelve una derivación por la izquierda de ω en otro caso devuelve un error.
También se puede obtener el *parse tree*.

Iniciar 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 mete a $Y_k...Y_1$ donde Y_1 es el nuevo tope;
>   se construye el nodo correspondiente en el árbol o se registra la producción en $M[X, a]$ como parte de la derivación
> - $M[X, a]$ no contiene una producción entonces hay un error.

Análisis sintáctico

Ejemplo:Algoritmo para LL

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

input: $id + id * id$

Análisis sintáctico

Ejemplo:Algoritmo para LL

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

input: $id + id * id$

en la tabla, $ es el símbolo de fin de cadena o de archivo

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

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:

$E \rightarrow TE' \rightarrow FT'E' \rightarrow idT'E' \rightarrow id\varepsilon E' \rightarrow id + TE' \rightarrow id + FT'E' \rightarrow id + idT'E' \rightarrow$

$$id + id * FT'E' \rightarrow id + id * idT'E' \rightarrow id + id + *id\varepsilon \rightarrow id + id + *id\varepsilon\varepsilon = id + id * id$$

Referencias

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman.
Compilers, Principles, Techniques and Tools.
Pearson Education Inc., Second edition, 2007.

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

[3] M. L. Scott.
Programming Language Pragmatics.
Morgan-Kauffman Publishers, Third edition, 2009.

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

[5] L. Torczon and K. Cooper.
Engineering A Compiler.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[6] F. A. Turbak and D. K. Gifford.
Design Concepts in Programming Languages.
The MIT Press, 2008.

Ejemplo e imágenes tomadas de libro “Compilers, Principles, Techniques and Tools”, capítulo 4.