Análisis Sintáctico: *Parser top-down*
Compiladores 2020-1, Nota de clase 4

Lourdes Del Carmen González Huesca Alejandra Krystel Coloapa Díaz

10 de septiembre de 2019
Facultad de Ciencias UNAM

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.

Analizador LL

El parsing recursivo descendente es un método top-down en el cual un conjunto de procedimientos recursivos es usado para procesar el programa. Usualmente se realiza un retroceso o backtracking para encontrar la producción correcta. Cada uno de ellos está relacionado con cada símbolo no-terminal de la gramática.

Un caso particular es el parsing predictivo que analiza un número fijo k de símbolos subsecuentes por adelantado. 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 (el siguiente token).

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

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

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

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 $X \to \alpha$:

1. Para cada terminal a en $\text{FIRST}(X)$ 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 $M[X, \#] = X \to \alpha$

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 incial contiene al símbolo inicial S. El algoritmo realiza una derivación izquierda, si w es la cadena que se ha reconocido hasta el momento, la pila contiene una secuencia de símbolos α tal que $S \to^* w\alpha$.

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.

```
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;
     }
```

Mientras X 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 $X \to Y_1..Y_k$, 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 en el árbol de derivación
  - $M[X, a]$ no contiene una producción entonces hay un error

Manejo de errores

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

Modo pánico Si dado el símbolo no terminal X, la entrada $M[X, a]$ está vacía, el modo pánico consiste en descartar tokens hasta que uno pertenezca al conjunto $SYNCH(X)$. La decisión de qué elementos pertenecen a $SYNCH(X)$ puede guiarse por las siguientes estrategias:

1. Incluir elementos en 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 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 sacarlo de la pila 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	$[T, E']$	$n + *n\#$	
ε	$[F, E']$	$n + *n\#$	
ε	$[n, E']$	$n + *n\#$	
n	$[E']$	$+ * n\#$	
n	$[+T]$	$+ * n\#$	
$n+$	$[T]$	$*n\#$	error
			T no deriva a $*$ y no pertenece ni a FOLLOWni FIRST
			descartamos el token con error "Simbolo * inesperado"
$n+$	$[T]$	$n\#$	n está en el FIRST(T), resumimos
$n+$	$[F]$	$n\#$	
$n+$	$[n]$	$n\#$	
$n + n$	[]	$\#$	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 la pila y/o la entrada para corregir el estado actual y resumir 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*	
...			
$n+$	$[T]$	$*n\#$	*error* : T esperaba un identificador, por lo que $M[T, *]$ crea un identificador "Falto un operador"
$n+$	$[T]$	$n*n\#$	resume análisis
$n+$	$[FT']$	$n*n\#$	
$n+$	$[nT']$	$n*n\#$	
$n+n$	$[T']$	$*n\#$	
$n+n$	$[*F]$	$*n\#$	
$n+n*$	$[F]$	$n\#$	
$n+n*$	$[n]$	$n\#$	
$n+n*n$	$[]$	$\#$	

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

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

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

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

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