Análisis Sintáctico
Compiladores 2020-1, Nota de clase 3

Lourdes Del Carmen González Huesca

27 de agosto de 2019
Facultad de Ciencias UNAM

La segunda fase de un compilador es el análisis sintáctico o *parsing* [1] Esta fase utiliza el resultado del analizador léxico, es decir la secuencia de tokens obtenidos del archivo fuente (sólo el nombre de ellos), para crear una estructura concreta que represente la forma gramatical del programa. Los lenguajes de programación son definidos formalmente a través de gramáticas libres de contexto. El objetivo de esta fase es reconocer palabras y verificar que la secuencia de tokens pueda ser generada por la gramática del lenguaje.

El análisis de la cadena de tokens siempre se realiza de izquierda a derecha y revisando símbolos por adelantado para construir el árbol de sintaxis concreta. El análisis del programa se lleva a cabo seleccionando una regla de producción de la gramática que coincida con el símbolo de lectura en turno y haciendo coincidir la producción son los símbolos subsecuentes. La representación del programa generada por el *parser* es un árbol de sintaxis concreta o ***parse tree*** cuyos nodos son símbolos no-terminales de la gramática y las hojas son símbolos terminales [2]. Si el programa no es reconocido como una secuencia correcta de palabras, según la gramática que define al lenguaje, entonces se deben reportar los errores sintácticos.

Gramáticas libres de contexto

Las gramáticas libres de contexto definen lenguajes al establecer las reglas para estructurar palabras. El lenguaje generado por una gramática se define como todas las derivaciones posibles o todos los árboles de derivación posibles. Lo anterior debe excluir a gramáticas que permitan dos o más derivaciones para una misma palabra. Entonces las gramáticas a considerar son aquellas que no son recursivas, no tienen transiciones épsilon y lo más importante: no son ambiguas.

Definición 1. Una gramática se dice **ambigua** si existe una palabra w con dos o más árboles de derivación distintos. En general una palabra puede tener más de una derivación, pero un sólo árbol y en tal caso no hay ambigüedad.
Un lenguaje L es ambiguo si existe una gramática ambigua G que genera a L y decimos que un lenguaje es **inherentemente** ambiguo si *todas* las gramáticas que lo generan son ambiguas.

Para eliminar la ambigüedad de una gramática no existe un algoritmo pero hay algunas maneras de reformular las reglas de producción de la gramática para eliminarla, por ejemplo se puede considerar la precedencia de algunos símbolos o el órden en que se puedne asociar algunos símbolos. La ambigüedad es sinónimo de no-determinismo, es decir que al procesar una cadena de símbolos se pueden obtener dos o más derivaciones o árboles. Esto se traduce en una indecisón sobre la estructura o forma del programa a analizar que es inaceptable.

[1] Cuya traducción al español es simplemente la acción de analizar sintácticamente un enunciado.
[2] A diferencia de un árbol de sintaxis abstracta o *abstract syntax tree* cuyos nodos y hojas son símbolos terminales.

Consideraremos las transformaciones de gramáticas que eliminan ϵ-producciones y las que son recursivas por la izquierda.

Definición 2. Una variable A se llama **anulable** si $A \to^\star \epsilon$, es decir si una derivación que empieza en A genera la cadena vacía.
Se da un algoritmo para hallar variables anulables:

Iniciar el conjunto *Anul* con las variables que tienen ϵ como producción

$$Anul := \{A \in V \mid A \to \epsilon \ \in P\}$$

Repetir la incorporación de variables que tienen producciones cadenas de variables anulables

$$Anul := Anul \cup \{A \in V \mid \exists\ A \to w \ \in P,\ \ w \in Anul^\star\}$$

Hasta que no se añaden nuevas variables a *Anul*

Una vez que se han identificado las variables anulables, la siguiente transformación de una gramática libre de contexto elimina exactamente las $\square$-producciones:

Para cada producción en la gramática que tenga la forma $A \to w_1 \dots w_n$ se deben agregar las producciones $A \to v_1 \dots v_n$ que son resultantes de los cambios de símbolos donde:

- $v_i = w_i$ si $w_i \notin Anul$, se respetan las variables no anulables
- $v_i = w_i$ ó $v_i = \epsilon$ si $w_i \in Anul$, las variables anulables pueden dejarse o eliminarse

Verificando que no se anulen todos los v_i al mismo tiempo.

Es decir, se van a respetar las producciones existentes y si alguna contiene las variables anulables se agregarán las producciones que resulten de eliminar las variables anulables en esa producción. Las ϵ-producciones desaparecerán.

Definición 3. Una producción recursiva por la izquierda es de la forma $A \to Aw$ con $w \in (V \cup T)^*$.
Ellas serán reemplazadas por producciones cuya recursión es por la derecha de la siguiente forma, para cada variable o símbolo no-terminal A crear dos categorías de reglas:

1. $A_{izq} = \{A \to Au_i \in P \mid u_i \in (V \cup T)^*\}$
2. $A_{der} = \{A \to v_j \in P \mid v_j \in (V \cup T)^*\}$

A continuación se obtendrán las reglas para la variable A como sigue:

$$A \to v_1 \mid \cdots \mid v_m \mid v_1 Z \mid \cdots \mid v_n Z$$

Además se incorporarán las siguientes reglas a P:

$$Z \to u_1 Z \mid \cdots \mid u_n Z \mid u_1 \mid \cdots \mid u_n$$

Ejemplo 1 (Gramática para expresiones aritméticas)**.** Consideremos la gramática

$$E \to E + E \mid E - E \mid E * E \mid E/E \mid (E) \mid n$$

donde n son los números naturales. Esta gramática es ambigüa ya que se pueden obtener diferentes árboles de sintaxis concreta para una misma cadena, por ejemplo:

Se puede transformar la gramática anterior en gramáticas equivalentes que no sean ambigüas [3]:

[3] Se pueden consultar referencias de cursos de Autómatas y Lenguajes Formales para revisar métodos para transformar gramáticas libres de contexto.

1. Gramática recursiva por la izquierda no-ambigüa

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T/F \mid F \\ F &\rightarrow (E) \mid n \end{aligned}$$

2. Eliminación de la recursión por la izquierda

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid -TE' \mid +T \mid -T \\ T &\rightarrow FT' \mid F \\ T' &\rightarrow *FT' \mid /FT' \mid *F \mid /F \\ F &\rightarrow (E) \mid n \end{aligned}$$

Observaciones Para cada gramática libre de contexto existe un parser que toma $O(n^3)$ en analizar una cadena de n símbolos terminales. Para lenguajes de programacón se pueden obtener analizadores más rápidos y en la práctica es posible obtener uno que tome tiempo lineal.

Parsers

Los métodos de análisis, generalmente se clasifican en dos:

- Analizadores *top-down* que construyen el árbol desde la raíz y hacia las hojas.
- Analizadores *bottom-up* que construyen el árbol desde las hojas hacia la raíz.

aunque también existen los analizadores universales para cualquier tipo de gramática.

Funciones auxiliares

Se definen a continuacón dos funciones auxiliares a usarse en los algoritmos de los parsers. Estas funciones consideran los símbolos terminales y no-terminales del lenguaje y permiten escoger una producción al considerar los siguientes símbolos en la cadena de entrada.

Definición 4 (Función First)**.** Esta función calcula el conjunto de símbolos terminales que están al inicio de las palabras derivadas de una cadena:

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

Recursivamente para un símbolo se tiene:

- Si $X \in \Sigma$ entonces $\textsc{First}(X) = \{X\}$.
- Si X es no-terminal y $X \rightarrow Y_1 Y_2 \dots Y_k$ es una producción con $k \geq 1$, entonces si $a \in \textsc{First}(Y_1)$ se tiene $a \in \textsc{First}(X)$.
  Es decir que $\textsc{First}(Y_1) \subseteq \textsc{First}(X)$.

Esta definición se puede generalizar a una cadena.

Definición 5 (Función FOLLOW)**.** La función FOLLOW para un símbolo no-terminal X calcula el conjunto de símbolos terminales que aparecen en una derivación cualquiera justo despúes de X:

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

Para esta función se agrega un nuevo símbolo para indicar el final del archivo a procesar mediante #. Este símbolo será útil en el analizador. Recursivamente:

- $\# \in \text{FOLLOW}(S)$ con S el símbolo inicial de la gramática.
- Si $A \to uXw$ entonces todo símbolo en $\text{FIRST}(w)$ está en $\text{FOLLOW}(X)$.
- Si existen producciones $A \to \alpha X$ o $A \to \alpha X \beta$ entonces $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(X)$.

Ejemplo 2 (Cálculo de los conjuntos FIRST y FOLLOW)**.** Consideremos la grámática para expresiones aritméticas definida en el Ejemplo 1. Los resultados de las funciones auxiliares son:

	E	E'	T	T'	F	S
FIRST	$\{(, n\}$	$\{+, -\}$	$\{(, n\}$	$\{*, /\}$	$\{(, n\}$	$-$
FOLLOW	$\{\#,)\}$	$\{\#,)\}$	$\{+, -, \#,)\}$	$\{+, -, \#,)\}$	$\{*, /, +, -, \#,)\}$	$\{\#\}$

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/`.