Conceptos de Lenguajes de Programación

Alberto Pardo Marcos Viera

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

Sintaxis

- En lenguajes de computadores, tanto la sintaxis como la semántica se deben especificar sin ambigüedades
  - Notaciones formales
- Distinguir entre sintaxis y semántica es de utilidad
- Sintaxis
  - Cómo especificar reglas estructurales del lenguaje: expresiones regulares, gramáticas libres de contexto
  - Cómo el compilador identifica la estructura: scanners, parsers

Tokens

Los **tokens** son los componentes básicos de un programa, los strings más cortos con significado individual.

- Ej. Palabras reservadas, identificadores, símbolos, constantes, etc.

Los tokens se especifican con **expresiones regulares**:

- Un carácter
- El string vacío (ϵ)
- Concatenación de dos expresiones regulares
- Alternativa entre dos expresiones regulares ($|$)
- Estrella de Kleene ($*$)

Ejemplo de Expresión Regular

$$
\begin{aligned}
\textit{number} &\longrightarrow \textit{integer} \mid \textit{real} \\
\textit{integer} &\longrightarrow \textit{digit}\ \ \textit{digit}* \\
\textit{real} &\longrightarrow \textit{integer}\ \ \textit{exponent} \mid \textit{decimal}\ \ (\textit{exponent} \mid \epsilon) \\
\textit{decimal} &\longrightarrow \textit{digit}*\ (\ .\ \ \textit{digit} \mid \textit{digit}\ \ .\ )\ \textit{digit}* \\
\textit{exponent} &\longrightarrow (\texttt{e} \mid \texttt{E})\ (\texttt{+} \mid \texttt{-} \mid \epsilon)\ \textit{integer} \\
\textit{digit} &\longrightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{aligned}
$$

Gramáticas Libres de Contexto

La notación de las **gramáticas libres de contexto** (CFG) suele llamarse Backus-Naur Form (BNF)

- Un conjunto de terminales (T)
- Un conjunto de no-terminales (N)
- Un no-terminal de inicio (S)
- Un conjunto de producciones, que pueden ser recursivas

EBNF si agregamos estrella y suma de Kleene.
Ejemplo de CFG:

$$
\begin{aligned}
expr &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ expr \mid \texttt{(}\ expr\ \texttt{)} \\
&\quad \mid expr\ op\ expr \\
op &\longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

CFG - Ejemplo mejorado

1. *expr* ⟶ *term* | *expr* *add_op* *term*
2. *term* ⟶ *factor* | *term* *mult_op* *factor*
3. *factor* ⟶ `id` | `number` | `-` *factor* | `(` *expr* `)`
4. *add_op* ⟶ `+` | `-`
5. *mult_op* ⟶ `*` | `/`

Análisis Sintáctico: Scanner

El **scanner** es usualmente el encargado de:

- reconocer los tokens
- quitar comentarios
- tratar los pragmas
- guardar el texto de los identificadores, números, strings
- guardar localizaciones en el fuente (archivo, línea, columna), para mensajes de error

Tres formas de construirlos:

- ad-hoc
  - Generalmente más rápidos y compactos
- Construir un DFA de forma semi-automática
  - Usualmente como cases anidados
- Construir un DFA dirigido por una tabla
  - Generadores como lex (flex) y scangen

Análisis Sintáctico: Parser

El **parser** es usualmente el encargado de:

- llamar al scanner para obtener los tokens del programa
- ensamblar los tokens en un árbol de sintaxis
- pasar el árbol a las siguientes fases del compilador

Mientras una CFG es una generadora de un lenguaje CF, un parser es un reconocedor del lenguaje.

Para cualquier CFG se puede crear un parser $O(n^3)$

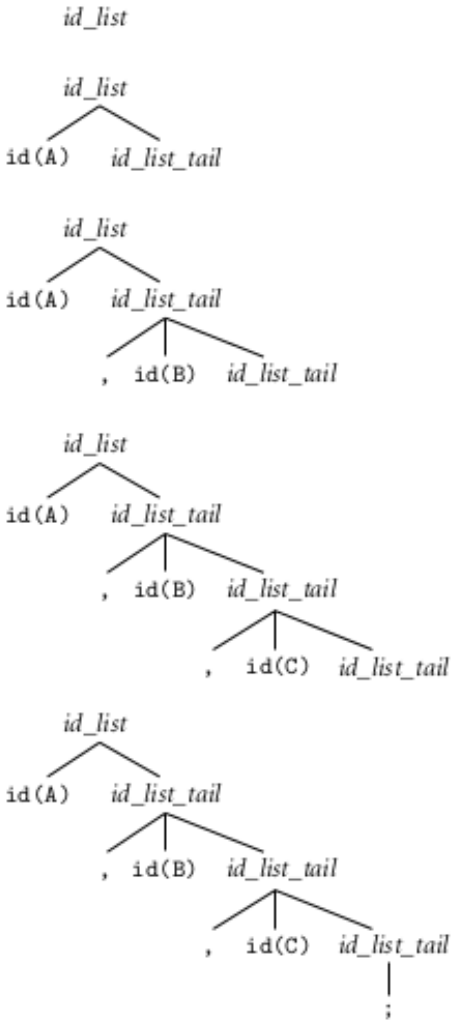
Existen clases de gramáticas para las que podemos construir parsers lineales. Las más importantes:

- **LL**: left-to-right scanning, left-most derivation
- **LR**: left-to-right scanning, right-most derivation

Parsers LL y LR

- Parsers LL se suelen llamar top-down o predictivos. Construyen el árbol desde la raíz, prediciendo en cada paso la siguiente producción a usar.
- Parser LR se suelen llamar bottom-up. Construyen el árbol desde las hojas, reconociendo cuando una colección de sub-árboles hijos se pueden unir con un padre.

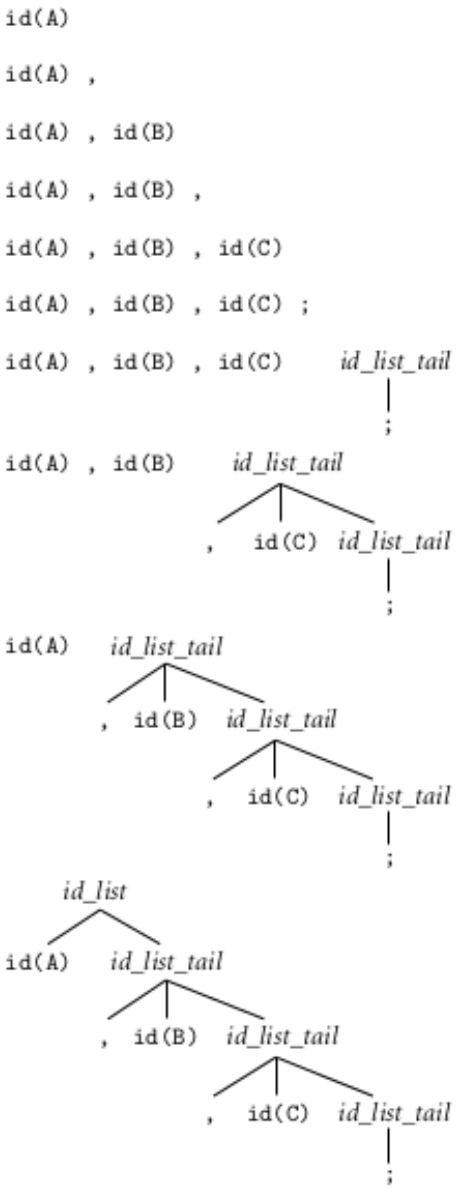
Parsers LL y LR



$id\_list \longrightarrow$ `id` id_list_tail

$id\_list\_tail \longrightarrow$ `,` `id` id_list_tail

$id\_list\_tail \longrightarrow$ `;`



Gramática LL(1)

program ⟶ *stmt_list* `$$`

stmt_list ⟶ *stmt* *stmt_list* | ϵ

stmt ⟶ `id :=` *expr* | `read id` | `write` *expr*

expr ⟶ *term* *term_tail*

term_tail ⟶ *add_op* *term* *term_tail* | ϵ

term ⟶ *factor* *factor_tail*

factor_tail ⟶ *mult_op* *factor* *factor_tail* | ϵ

factor ⟶ `(` *expr* `)` | `id` | `number`

add_op ⟶ `+` | `-`

mult_op ⟶ `*` | `/`

- Puedo construir un parser recursivo descendente (o generarlo con ANTLR)
- O generar un LL parse table con un driver program.

LR

- Casi siempre se basan en tablas.
- Mantiene un stack con las raíces de los subárboles parcialmente completados.
- shifts tokens al stack
- reduces símbolos en tope del stack