Universidad del Valle de Guatemala
Teoría de la Computación, CC2019
Docentes: Alan Reyes; Tomás Gálvez P.
Semestre 2, 2022

UVG
UNIVERSIDAD DEL VALLE DE GUATEMALA

Sintaxis e introducción a las gramáticas libres de contexto

De acuerdo con Wikipedia en 2022: "La sintaxis es la parte de la gramática que estudia las reglas y principios que gobiernan la combinatoria de constituyentes sintácticos y la formación de unidades superiores a estos, como los sintagmas y las oraciones gramaticales.". Los constituyentes sintácticos son componentes de las oraciones definidos de forma recursiva a partir de una o más palabras siguiendo las reglas de la sintaxis. Esto quiere decir que un constituyente sintáctico puede estar conformado por otros constituyentes sintácticos y formar parte de otro constituyente superior.

En sistemas matemáticos y en el contexto de compiladores, la sintaxis es expresada comúnmente usando la notación **BNF** (*Backus-Naur Form*, introducida por John Backus de IBM para describir la sintaxis de ALGOL, y, más adelante, simplificada por Peter Naur). Esta notación describe conjuntos de reglas gramaticales, denominados **gramáticas**.

Anteriormente mencionamos los lenguajes regulares como el conjunto de cadenas formadas a partir de una expresión regular. Con listados de expresiones regulares construimos definiciones regulares que nos permitían "ampliar" nuestro conjunto de símbolos para conformar expresiones regulares cada vez más complejas. Nuestros ejemplos de definiciones regulares se plantearon usando notación BNF, y con las gramáticas definimos lenguajes de manera similar, pero eliminando una importante restricción: la imposibilidad de definiciones recursivas.

Con esto en mente observamos que las gramáticas permiten construir lenguajes como lo hacen las definiciones regulares, pero su capacidad de expresión es mayor (*i.e.*, definen lenguajes más amplios y complejos que los que puede describir una definición regular). Nos concentramos en un tipo específico de gramáticas, llamadas **gramáticas de tipo 2** o **gramáticas libres de contexto.** Una gramática libre de contexto posee lo siguiente:

- Un conjunto de símbolos **terminales**, definidos por la gramática como elementos inmediatamente reconocibles e indivisibles (*e.g.*, palabras reservadas o el *string* vacío ε).
- Un conjunto de símbolos **no-terminales**: símbolos compuestos que representan una estructura sintáctica. Su composición se especifica con producciones, definidas a continuación.
- Un conjunto de **producciones**, que son reglas conformadas por (de izquierda a derecha) un símbolo no-terminal, una flecha ($\rightarrow$) y una secuencia de símbolos terminales y/o no-terminales. El término del lado izquierdo de la flecha es la **cabeza** de la producción, mientras que lo que está en el lado derecho es el **cuerpo**. Puesto que se pueden definir varias producciones con una misma cabeza, es normal agrupar éstas en una única producción con diferentes cuerpos separados por el símbolo "|", que se lee como "*or*" (similar a como hicimos con las definiciones regulares).
- Un no-terminal designado como el **símbolo de inicio**, regularmente identificado por estar a la cabeza de la primera producción en la gramática.

Dada una cadena de símbolos (que representaría, por ejemplo, un programa fuente) el procedimiento de *parseo* busca reconocer el símbolo de inicio de la gramática del lenguaje. Para ello procesa el cuerpo de la producción correspondiente, identificando los símbolos terminales que encuentre en la cadena y apoyándose en las producciones de los símbolos no-terminales para el mismo propósito.

Ejemplo de una gramática libre de contexto expresada con BNF:

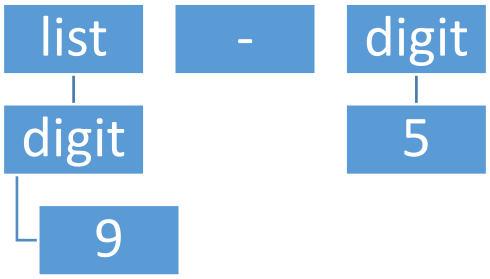
$$list \rightarrow list + digit$$

$$list \rightarrow list - digit$$

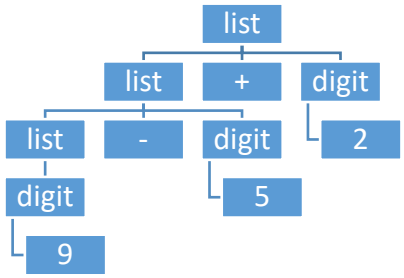
$$list \rightarrow digit$$

$$digit \rightarrow 0|1|2|3|4|5|6|7|8|9$$

Esta gramática describe *strings* de dígitos separados por signos de suma y resta. Por lo tanto, una cadena descrita por esta gramática sería 9-5+2. ¿Cómo? Es fácil observar que 9, 5 y 2 son dígitos, pero la producción que describe una resta no consta de dos dígitos sino de un dígito y una $list$. Para adaptarlo aprovechamos la producción que dice que una $list$ también puede ser un dígito.



Lo que está en el nivel más alto en el diagrama representa la estructura de la primera operación binaria de acuerdo con la gramática, que es el cuerpo de una $list$. Cuando leemos el siguiente caracter, dado que es un operador de suma, reconocemos automáticamente que estamos en la primera producción de la gramática (ya tenemos la $list$ del lado izquierdo de la operación), entonces estaríamos esperando un dígito a continuación para completar el cuerpo de $list$ descrito por una suma. Efectivamente vemos que 2 es un dígito, entonces podemos armar el árbol así:



Lo que acabamos de producir es un **árbol sintáctico**, una representación de la cadena que provee información sobre la estructura de la misma. El proceso de obtener el árbol sintáctico, también descrito como la **derivación** de la cadena a partir de un símbolo de inicio, es lo que denominamos como *parseo*.

Ejemplificado aquí, y como regla general, los árboles sintácticos tienen en la raíz al símbolo de inicio de la gramática; en sus nodos internos tienen no-terminales; y en sus hojas tienen símbolos terminales únicamente. Falta a cualquiera de estas reglas produce un error de sintaxis. ¿Cuál es el alfabeto de esta gramática? Simplemente hay que ver los símbolos terminales en ella, que serían los dígitos del 0 al 9 y los operadores $+$ y $-$.

La razón por la que estas gramáticas son llamadas *libres de contexto* es porque la forma que tienen las producciones está restringida a un único no-terminal a la cabeza. Esto permite reemplazar dicho no-terminal por el cuerpo de su producción en una cadena de símbolos, sin importar lo que el no-terminal tenga a su alrededor. En las gramáticas **sensibles al contexto** las producciones pueden tener la forma $\alpha A \beta \rightarrow \alpha \gamma \beta$ donde α y β son cadenas de terminales y/o no-terminales, y γ es también una cadena de terminales y/o no-terminales pero que no puede estar vacía.

Como ejemplo, propongamos una sintaxis que describa las llamadas a métodos en Java. Supongamos, por conveniencia, que los identificadores son terminales, y evitemos describir en qué consiste un parámetro, limitándonos a llamarlo $param$.

Una llamada a métodos en Java se realiza con el nombre del método seguido de la lista de argumentos, separados por comas, dentro de paréntesis. *E.g.*: `myMethod(a, b, c)`. Dado que `myMethod` es un nombre decidido por el programador o programadora, lo llamamos un **identificador** y le asignamos el símbolo id. También suponemos que los paréntesis son terminales ya que no puede haber otro símbolo en su lugar. Lo que está entre ellos es una lista de parámetros separada por comas. Como vimos en la gramática anterior, la forma de describir una lista se hace como en la producción que describe una suma (o resta) pero reemplazando el símbolo '+' por ',':

$$params \rightarrow params, param$$

Hay que observar que una llamada a método puede tener solo un parámetro, entonces también deberíamos incluir una producción que permita que nuestra lista $params$ consista únicamente de un parámetro:

$$params \rightarrow params, param$$

$$params \rightarrow param$$

Para abreviar, juntamos ambas producciones en una de la siguiente manera:

$$params \rightarrow params, param \mid param$$

También podemos tener una llamada a método sin parámetros. Para representar esa ausencia de elementos usamos ε.

Ejercicio de práctica A: no sería correcto permitir que $params$ o $param$ tuvieran una producción con cuerpo ε. ¿Por qué?

Sabemos que nuestro método debe ser llamado con una lista de uno o más parámetros, o con nada, entonces podemos introducir una producción para esto:

$$optparams \rightarrow params \mid \varepsilon$$

Por último, ya que teníamos descrito en qué consiste una llamada a método, sólo necesitamos describirlo en notación BNF:

$$call \rightarrow id(optparams)$$

Nuestra gramática quedaría así:

Universidad del Valle de Guatemala
Teoría de la Computación, CC2019
Docentes: Alan Reyes; Tomás Gálvez P.
Semestre 2, 2022

$$call \rightarrow id(optparams)$$

$$optparams \rightarrow params \mid \varepsilon$$

$$params \rightarrow params, param \mid param$$

Ejercicio de práctica B: considere la gramática de abajo.

$Stm \rightarrow Stm; Stm$	(CompoundStm)	$ExpList \rightarrow Exp, ExpList$	(PairExpList)
$Stm \rightarrow id := Exp$	(AssignStm)	$ExpList \rightarrow Exp$	(LastExpList)
$Stm \rightarrow \text{print}(ExpList)$	(PrintStm)	$Binop \rightarrow +$	(Plus)
$Exp \rightarrow \text{id}$	(IdExp)	$Binop \rightarrow -$	(Minus)
$Exp \rightarrow \text{num}$	(NumExp)	$Binop \rightarrow \times$	(Times)
$Exp \rightarrow Exp\ Binop\ Exp$	(OpExp)	$Binop \rightarrow /$	(Div)
$Exp \rightarrow (Stm, Exp)$	(EseqExp)		

Stm representa un *statement*, *Exp* una expresión y *ExpList* una lista de expresiones. *Parsee* el siguiente programa (es decir, construya el árbol sintáctico siguiendo las reglas de la gramática) apoyándose en los nombres de cada producción para diferenciar la ocurrencia de cada símbolo no-terminal en el árbol; y priorizando el reconocimiento de símbolos de izquierda a derecha:

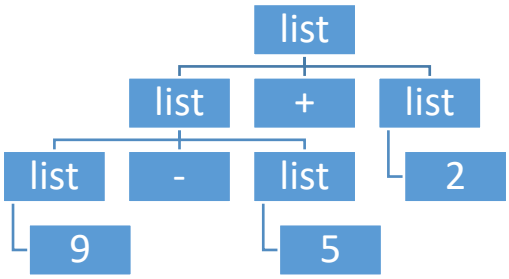
```
a := 5 + 3; b := ( print( a, a - 1 ), 10 * a ); print(b)
```

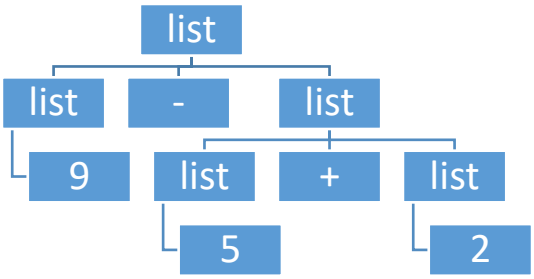
¿Qué problema percibe con esta gramática al realizar este *parseo*?

En el ejemplo de la gramática de dígitos y operaciones aritméticas ¿qué pasaría si evitamos el no-terminal *digit* y nos saltamos de una vez a los dígitos de una lista como en la siguiente producción?

$$list \rightarrow 0|1|2|3|4|5|6|7|8|9$$

El resultado sería que, en las producciones de suma y resta, necesitaríamos que los no-terminales a cada lado los operadores fueran *list*. En ese caso podríamos armar dos árboles sintácticos diferentes para el programa 9-5+2: el que obtuvimos en el ejemplo y uno donde la suma 5+2 sea realizada de primero.





Dado que el significado (resultado) de cada árbol sintáctico es distinto del otro, nuestra gramática es **ambigua**; un problema que debemos evitar.

En algunos casos, gramáticas ambiguas pueden ser transformadas en gramáticas no ambiguas por medio de técnicas como la asociación por lado. En el ejemplo anterior pasamos de una gramática no ambigua a una ambigua ignorando las producciones que permitían distinguir los lados de una operación de suma o resta. De acuerdo con las convenciones regulares, nuestra gramática no-ambigua era **asociativa a la izquierda** porque lo que está a la izquierda del operador debe procesarse antes (considerando que son solo sumas y restas). Si hubiéramos querido que la asociatividad se diera hacia la derecha (útil en el caso de operadores como la potencia) podríamos haber intercambiado los lados de las producciones para obtener:

$$list \rightarrow digit + list$$

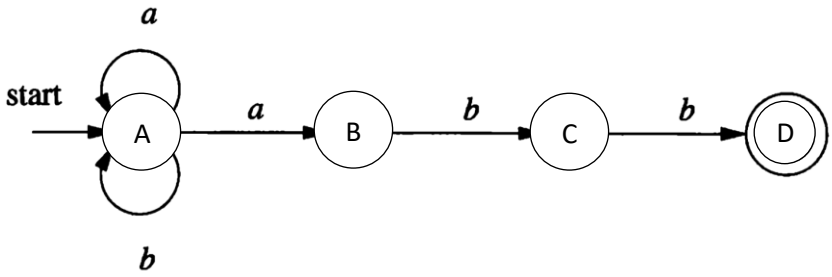
$$list \rightarrow digit - list$$

En estos casos, puesto que las producciones de $digit$ tienen únicamente terminales en sus cuerpos, sabemos que el primer término terminaría su análisis en un número nada más, mientras que el no-terminal $list$ del lado derecho del operador deja la oportunidad para que se sinteticen otras operaciones aritméticas antes de efectuar la operación que describe el cuerpo de la producción.

Ejercicio de práctica C: construya una gramática que describa la sintaxis de los números romanos hasta los millares.

Las gramáticas libres de contexto describen **lenguajes libres de contexto** y éstos tienen relación con los lenguajes regulares que hemos estado trabajando. Si tenemos dos lenguajes libres de contexto L_1 y L_2, los resultados de las operaciones $L_1 \cup L_2$, $L_1 L_2$ y ${L_1}^*$ son todos lenguajes libres de contexto. Además, los lenguajes regulares son un subconjunto de los lenguajes libres de contexto. Desde una perspectiva práctica esto significa que todo lenguaje regular puede ser descrito con una gramática libre de contexto, y en este caso lo que tendríamos es una **gramática regular**.

En las gramáticas regulares las producciones tienen la forma $A \rightarrow \sigma B$ o $A \rightarrow \varepsilon$, donde σ es un símbolo del alfabeto (terminal); y A y B son no-terminales. Si conocemos el autómata de un lenguaje regular podemos armar la correspondiente gramática regular haciendo de cada estado un no-terminal, armando una producción con ese no-terminal a la cabeza por cada transición que salga del estado, y definiendo el cuerpo de esta producción con el símbolo de transición seguido del no-terminal (estado) que se alcanza con esa transición. Por ejemplo:



Este autómata corresponde a la expresión regular $(a|b)^* abb$. La gramática que describe el mismo lenguaje que este autómata es:

$$A \rightarrow aA$$

$$A \rightarrow bA$$

$$A \rightarrow aB$$

$$B \rightarrow bC$$

$$C \rightarrow bD$$

$$D \rightarrow \varepsilon$$

D es el estado de aceptación al ponerlo a la cabeza de una producción cuyo cuerpo es ε. Si el autómata tuviera transiciones de salida en desde su estado de aceptación las manejaríamos igual, pero siempre habría una producción con el estado de aceptación a la cabeza y ε en su cuerpo.

Fuentes:

- https://es.wikipedia.org/wiki/Constituyente_sint%C3%A1ctico
- https://es.wikipedia.org/wiki/Sintaxis
- http://www.cs.cornell.edu/courses/cs412/2008sp/lectures/lec11.pdf
- http://en.wikipedia.org/wiki/Context-sensitive_grammar
- Aho, A. V., Lam, M. S., Ravi, S., & Ullman, J. D. (2007). *Compilers: Principles, Techniques and Tools.* Pearson.
- Appel, A. (2004). *Modern Compiler Implementation in (Java|C|ML).* Cambridge.
- Louden, K. C. (1997). *Compiler Construction: Principles and Practice.* Cengage Learning.
- Martin, J. C. (2003). *Introduction to Languages and the Theory of Computation.* McGraw-Hill.