Capítulo 2

Modelando con gramáticas

Este capítulo muestra ejemplos de definiciones de gramáticas a partir de definiciones informales de los lenguajes. Luego, describimos la extensión al formalismo de gramáticas conocido como BNF extendido. A partir de la descripción de un lenguaje con este formalismo se puede construir un programa que reconozca las cadenas del lenguaje: un compilador si/no. Es decir, un programa que recibe una cadena y puede decir si la cadena pertenece o no al lenguaje. El método que se describe en este capítulo es el de descenso recursivo. A pesar de que es relativamente fácil construir estos compiladores a partir de la gramática, existen herramientas que hacen esto automáticamente. Una de estas herramientas es JavaCC. En este capítulo se mostrará cómo usar JavaCC, un generador de compiladores para Java.

Usaremos las siguientes convenciones para la descripción de gramáticas: los símbolos no-terminales se escribirán en rojo, los terminales en azul, y los símbolos de la gramática como la flecha, lambda y los que se usan en el BNF-extendido en negro.

4.1 Definición de Gramáticas

En esta sección se presentan algunos ejemplos de definición de gramáticas a partir de descripciones en lenguaje natural. No se pueden dar unas reglas infalibles para este proceso, ya que en muchos casos depende mucho de la intuición del diseñador. Cada sub sección de esta sección corresponde a un ejemplo donde se ilustra el proceso de construcción de gramáticas para problemas propuestos. El lector notará que si se parte de una definición recursiva es muy fácil obtener la gramática correspondiente.

4.1.1 Expresiones-S

Las expresiones S son la base del lenguaje de programación LISP. Podemos definir un sub-conjunto de estas expresiones recursivamente así:

- Un entero es una expresión S
- Si $E_1, \ldots, E_n$ son expresiones-S (para $n > 0$), entonces las siguientes también son expresiones-S.
    1. $(+E_1 \ldots E_n)$
    2. $(*E_1 \ldots E_n)$

Primero se debe determinar cuáles son los símbolos terminales. En este caso, vemos que los terminales son los números (NUM), abrir paréntesis, cerrar paréntesis y los símbolos + y *.

Digamos que S es el símbolo distinguido para generar expresiones S. Del primer punto obtenemos:

$S \rightarrow NUM$

Para las siguientes reglas tenemos un problema ya que no podemos escribir algo como $E_1 \ldots E_n$. Lo que estamos diciendo en realidad es que tenemos una lista de una o más expresiones a las que se les antepone un operando. La regla es entonces:

$S \rightarrow (OP\ Ls)$

Donde Ls es una lista de una o más expresiones S. Una Lista de una o más expresiones-S es: una expresión-S o una Lista de expresiones-S seguida de una expresión-S. Eso nos da las siguientes dos reglas:

$Ls \rightarrow S$

$Ls \rightarrow Ls\ S$

Falta entonces definir OP:

$OP \rightarrow *$

$OP \rightarrow +$

De hecho, no es necesario definir OP aparte y la siguiente gramática es suficiente

$S \rightarrow NUM$

$S \rightarrow (+\ Ls)$

$S \rightarrow (*\ Ls)$

$Ls \rightarrow S$

$Ls \rightarrow Ls\ S$

4.1.2 Un lenguaje para un editor gráfico

- Un dibujo se puede representar como una secuencia no vacía de objetos, separados por comas, entre paréntesis y precedidos de la palabra "DIBUJO". Es decir, es una cadena de la forma **DIBUJO(*O*$_1$,…, *O*$_n$)**, donde cada ***O*$_i$** es una cadena que representa un objeto.
- Un objeto puede ser un círculo, un punto, una línea, un rectángulo, o un cuadro de dibujo especificados con cadenas de las formas descritas abajo:
  - **CIRC(*Coordenadas, Radio, Color*$_1$*, Color*$_2$**): Representa un círculo con centro en ***Coordenadas***, radio ***Radio***, borde de color ***Color*$_1$** y relleno de color ***Color*$_2$**.
  - **PUNT(*Coordenadas, Color*)**: Representa un punto localizado en ***Coordenadas*** de color ***Color***.

- **LINEA(*Coordenadas$_1$, Coordenadas$_2$, Color*):** Representa una línea que va de ***Coordenadas$_1$*** a ***Coordenadas$_2$*** y con color ***Color***.
- **RECT(*Coordenadas$_1$, Coordenadas$_2$, Color$_1$, Color$_2$*):** Representa un rectángulo con esquinas opuestas ***Coordenadas$_1$*** y ***Coordenadas$_2$***, borde de color ***Color$_1$*** y relleno de color ***Color$_2$***.
- **DIB(*Coordenadas$_1$, Coordenadas$_2$, Color$_1$, Color$_2$, (O_1, …,O_n)*):** Un cuadro de dibujo es un rectángulo que contiene dibujos. Se especifica igual que un rectángulo cambiando el nombre a DIB y agregando una lista de objetos entre paréntesis.

- ***Coordenadas*** es una pareja de la forma ***(X, Y)*** donde ***X*** y ***Y*** son enteros positivos.
- ***Color*** es una tripla ***(A, V, R)*** donde ***A***, ***V***, y ***R*** son números enteros positivos y negativos.
- ***Radio*** es un entero

Primero que todo, hay que determinar cuáles son los símbolos terminales. En este caso son los siguientes: los enteros (NUM), el símbolo -, para distinguir enteros negativos, abrir paréntesis, cerrar paréntesis, coma, y las palabras, DIBUJO, RECT, LINEA, PUNT, CIRC y DIB.

Queremos definir una gramática para un dibujo, entonces usamos D como símbolo distinguido y a partir del primer punto en la descripción del lenguaje tenemos:

D → DIBUJO(Os)

Los puntos suspensivos no hacen parte de una gramática, entonces no pueden usarse en las producciones; se debe definir una lista de uno o más objetos usando el no terminal Os.

Se define una lista de uno o más objetos recursivamente así: una lista es un objeto o es una lista seguida de una coma seguida de un objeto. Eso nos da las siguientes dos reglas:

Os → Ob

Os → Os , Ob

Ahora se dan las reglas para generar descripciones de objetos. Cada uno de los puntos de la definición dada, genera una regla:

Ob → CIRC(Coor , Radio , Color , Color)

Ob → PUNT(Coor , Color)

Ob → LINEA(Coor , Coor , Color)

Ob → RECT(Coor , Coor , Color , Color)

La definición de cuadro de dibujo también hace referencia a una lista de dibujos:

Ob → DIB(Coor , Coor , Color , Color , (Os))

Aún faltan las definiciones de Coordenadas, Radio y Color:

Coor → (NUM , NUM)

Radio → NUM

Como color se compone de números que pueden ser positivos o negativos, se agrega un no terminal para representar enteros:

Color → (Z , Z , Z)

Finalmente, esta es la definición de entero positivo o negativo:

Z → -NUM

Z → NUM

4.1.3 El lenguaje de las expresiones aritméticas

Las expresiones aritméticas son expresiones formadas con los operadores binarios suma, resta, multiplicación, división y el operador prefijo para el cambio de signo.

Una primera aproximación sería la siguiente gramática.

E → E + E

E → E - E

E → E * E

E → E / E

E → - E

E → num

E → (E)

Si se quiere que la gramática incorpore las propiedades de asociatividad por la izquierda y de la precedencia de operadores usual, se define la gramática así:

E → E + T

E → E - T

E → T

T → T * F

T → T / F

T → F

F → - P

F → P

P → num

P → (E)

El símbolo distinguido es E. Note que los operadores de suma y resta que tienen menos precedencia son los que aparecen primero en una derivación a partir del símbolo distinguido. Los paréntesis y el operador unario para cambiar de signo, operadores que tienen más precedencia, aparecen más lejos del símbolo distinguido.

4.1.4 El lenguaje de las gramáticas

Suponga que queremos definir una gramática para definir gramáticas independientes del contexto. Recuerden que según la descripción dada en el Capítulo 1, una gramática es una cuádrupla cuyo primer componente es un conjunto de símbolos no-terminales; el segundo componente es un conjunto de símbolos no terminales; el tercer componente es un símbolo distinguido; y el cuarto componente es un conjunto de producciones. Una producción es un no-terminal seguido por una flecha, seguido por el símbolo lambda o por una secuencia no vacía de terminales y no-terminales. En cada uno de los casos de arriba el conjunto se denota como una lista entre corchetes.

Para facilitar las cosas, suponga que tenemos un analizador léxico que nos reconoce los siguientes símbolos terminales:

{**term**, **noTerm**, "**(**", "**)**", "**λ**", "**,**", "**{**", "**}**"}.

Es importante notar que el símbolo "**λ**" acá es un terminal. Recuerden que la parte derecha de las reglas puede ser una secuencia de símbolos terminales y noterminales o la cadena vacía representada por el símbolo lambda.

Se comienza entonces con la definición:

- Una gramática es una cuádrupla…

  Gramatica → ({NTs}, {Ts}, noTerm, {Ps})

- NTs es una lista de no terminales separados por comas

  NTs → noTerm

  NTs → NTs , noTerm

- Ts es una lista de terminales separados por comas

Ts → term

Ts → Ts , term

- Ps es una lista de producciones separadas por comas

Ps → P

Ps → Ps , P

- Una producción es un no-terminal seguido por una flecha, seguido por el símbolo lambda o por una secuencia no vacía de terminales y no-terminales.

P → noTerm → ParteDerecha

ParteDerecha → λ

ParteDerecha → Sims

Sims → Sim

Sims → Sims Sim

Sim → term

Sim → noTerm

4.1.5 Un lenguaje para manejo de inventarios

Digamos que queremos verificar si un archivo de datos usado para agregar inventarios a un almacén tiene el formato correcto. Se describe informalmente el formato como sigue:

El archivo tiene una secuencia de comandos, uno por línea. Donde cada comando puede ser:

- **INSERTAR_NUEVO** *nombre codigo precio cantidad*
- **ACTUALIZAR CANTIDAD POR** *codigo num*
- **ACTUALIZAR CANTIDAD** *codigo num*
- **CAMBIAR PRECIO** *codigo precio*
- **CAMBIAR PRECIO POR** *codigo porcentaje*
- **PARA CADA CODIGO EN { *listadeCodigos* }**

  comandosdeActualizacion

  FINPARA

Los comandos de actualización pueden tener el siguiente formato:

- **ACTUALIZAR CANTIDAD POR** *num*
- **ACTUALIZAR CANTIDAD** *num*
- **CAMBIAR PRECIO** *precio*
- **CAMBIAR PRECIO POR** *porcentaje*

Lista de Códigos: Códigos separados por comas.

Suponga que ya se tiene un analizador léxico que reconoce los siguientes tokens (símbolos terminales):

- Código
- Precio
- Porcentaje
- Nombre
- Símbolo de fin de línea EOL
- Coma (separador en las listas) COMMA
- Abrir y cerrar corchete: OBR, CBR
- Además se tendría un token por cada una de las palabras reservadas
    - **INSERTAR_NUEVO**
    - **ACTUALIZAR**
    - **CANTIDAD**
    - **POR**
    - **CAMBIAR**
    - **PRECIO**
    - **PARA**
    - **CADA**
    - **CODIGO**
    - **EN**
    - **FINPARA**

Comenzamos a definir la gramática. En la descripción dice que el archivo es una secuencia de comandos, uno por línea. Suponga que el símbolo distinguido es Coms. En este caso, el fin del línea no es un separador. Cada comando debe tener un fin de línea al final.

1. COMs → COM EOL
2. COMs → COMs COM EOL

Pasamos a definir los comandos,

3. COM → INSERTAR_NUEVO nombre codigo precio cantidad
4. COM → ACTUALIZAR CANTIDAD POR codigo num
5. COM → ACTUALIZAR CANTIDAD codigo num
6. COM → ACTUALIZAR CANTIDAD POR codigo num
7. COM → CAMBIAR PRECIO POR codigo porcentaje
8. COM → CAMBIAR PRECIO codigo precio
9. COM → PARA CADA CODIGO EN OBR Codigos CBR EOL ListaCs FINPARA

En la última regla quedaron no terminales pendientes de definir la lista de códigos y la lista de comandos internos.

10. Codigos → OBR Cs CBR
11. Cs → Cs COMA Codigo
12. Cs → Codigo

13. ListaCs → COMi EOL
14. ListaCs → ListaCs COMi EOL

Los comandos internos se diferencian de los que ya se definieron en que no es necesario poner el código. Necesitamos nuevas reglas para estos.

15. COMi → ACTUALIZAR CANTIDAD num
16. COMi → ACTUALIZAR CANTIDAD POR num
17. COMi → CAMBIAR PRECIO POR porcentaje
18. COMi → CAMBIAR PRECIO precio

Ejercicios

1. Defina una gramática para las expresiones booleanas formadas con las constantes **true** y **false,** y los conectivos: $\Rightarrow$, $\Leftrightarrow$, $\wedge$, $\vee$, y $\neg$.
2. Defina una gramática para las expresiones regulares bien formadas sobre el alfabeto {a,b}.
3. El lenguaje Prolog es un lenguaje lógico. Los programas Prolog se definen como una secuencia de hechos y reglas seguidos por una consulta. Comenzamos por definir los constructos básicos del lenguaje que serán los terminales de la gramática que genera el lenguaje.
   a. Un átomo es un número o un nombre.
   b. Un nombre es una secuencia cadena alfanumérica que comienza con una letra minúscula.
   c. Una variable es una secuencia alfanumérica que comienza con una mayúscula.

   A continuación se definen los constructos compuestos.

   a. Un funtor es un nombre que puede o no estar seguido de una secuencia de argumentos separados por comas entre paréntesis.
   b. Un argumento es un funtor, un número o una variable.
   c. Un hecho es un funtor seguido por un punto (.).
   d. Una regla es un funtor seguido por el operador ":-" seguido por una secuencia de uno o más funtores separados por comas y seguidos por un punto.
   e. Una consulta es el operador "?-" seguido por una secuencia de uno o más funtores separados por comas.

Suponiendo que los terminales son los nombres, los enteros, las variables, la coma, el punto y los operadores: "?-" y ":-". Defina la gramática para un programa Prolog.

4. Defina la gramática para las declaraciones de variables en Java.
5. Defina la gramática para la definición de los encabezados de los métodos en Java.
6. Agregue las operaciones resta y división a la gramática de la sección 4.1.3
7. Agregue la operación prefija para cambiar signo a la gramática de la sección 4.1.3
8. Agregue la operación de elevar es decir: (expresión)^Entero a la gramática de la sección 4.1.3
9. Agregue la invocación a la función raíz cuadrada: sqrt a la gramática de la sección 4.1.3

10. Agregue el uso de variables, representadas con una letra mayúscula o minúscula: A-Z a la gramática de la sección 4.3. Puede suponer que los nombres de variables son terminales.

11. Enriquezca la gramática para Prolog con la siguiente regla: "una argumento también puede ser una lista.Una lista se define así:

    - La lista vacía denotada **[ ]** es una lista
    - Si $\mathbf{a_1, ..., a_n}$ son argumentos y **L** es una lista ENTONCES LAS SIGUIENTES TAMBIÉN SON LISTAS:
        - $\mathbf{[a_1, ... , a_n \mid L]}$
        - $\mathbf{[a_1, ... , a_n]}$

4.2 Extendiendo el formalismo

Las gramáticas independientes del contexto pueden extenderse para simplificar la descripción de ciertos lenguajes. Esto no aumenta su expresividad. Estas gramáticas se llaman gramáticas BNF – extendidas (Bakus Naur Form) en honor de J. Bakus y P.Naur quienes usaron este formalismo para la descripción de Algol[1].

En esta sección describimos el formalismo y convertimos algunas de las gramáticas de la sección anterior a BNF extendido. La sección 4.2.1 describe el formalismo y la sección 4.2.2 muestra algunos ejemplos.

4.2.1 Descripción

Para distinguir los símbolos terminales de los no-terminales, los no terminales se escribían dentro de corchetes triangulares y en lugar de flecha usaban "::=". Esto era necesario ya que no se tenían conjuntos de caracteres extendidos y facilitaba la impresión. En estas notas no vamos a usar estas convenciones. En lugar de esto, usaremos colores para distinguir los símbolos terminales de los no terminales. Los no terminales los escribimos en rojo, los terminales en azul y los símbolos de gramática en negro.

Tal como su nombre lo indica, el BNF extendido, extiende la notación de las gramáticas independientes del contexto para facilitar la notación; no extienden la expresividad del formalismo. Es más bien un azúcar sintáctico. Las reglas siguen siendo de la formal: NoTerminal $\rightarrow$ ParteDerecha.

La parte derecha de las reglas es de la forma: $\beta_1\beta_2\ldots\beta_n$, donde Cada β_i es una expresión que puede ser un terminal, un no-terminal, o de la forma: $\mathbf{\{}\delta_1\delta_2\ldots\ \delta_k\mathbf{\}}$, $\mathbf{[}\delta_1\delta_2\ldots\ \delta_s\mathbf{]}$, o $\mathbf{(}\delta_1|\delta_2|\ldots\ |\delta_n\mathbf{)}$ donde cada δ_i es a su

[1] Para un resumen de la historia de los lenguajes de programación se recomienda la introducción a [PRATT1996]

vez una expresión. La extensión se refiere a las tres últimas formas que describimos abajo.

- $[\delta_1\delta_2\ldots\ \delta_s]$: Esto quiere decir que los símbolos que se encuentran entre los corchetes cuadrados son opcionales.
- $\{\delta_1\delta_2\ldots\ \delta_k\}$: Esto quiere decir que la cadena que está entre los corchetes se puede repetir cero o más veces.
- (cad1 | cad2): Tiene la misma semántica que en las expresiones regulares.
- Además se usan paréntesis para agrupar.

Dado que tenemos la posibilidad de poner opciones en cada regla, podemos restringir a que haya una sola regla por cada no terminal.

Es claro que siempre que tenemos una gramática en formato BNF extendido, la podríamos cambiar a una gramática sin las extensiones.

El problema que tenemos, entonces, es convertir una gramática independiente del contexto a un BNF extendido. Para esto tenemos las reglas que aparecen en la Tabla 4.1.

Note que si sólo aplicamos la primera regla, ya tendríamos una gramática en formato BNF extendido. Sin embargo, a veces es útil aplicar las otras reglas para disminuir el número de reglas de una gramática. Esto no necesariamente la hace más entendible pero facilita la construcción de analizadores sintácticos a partir de las gramáticas. Esto se discutirá en la siguiente sección.

A continuación se mostrará cómo se puede convertir una gramática independiente del contexto a una gramática en formato BNF extendido que haga uso de todas las extensiones mencionadas.

Regla de Transformación	Si tiene:	Cambie a:
1	• $A\rightarrow \beta_1$ • $A\rightarrow \beta_2$ • … • $A\rightarrow \beta_n$	$A\rightarrow \beta_1 \mid \beta_2 \mid\ldots\mid \beta_n$
2 Ley de arden	$A\rightarrow(A\varphi)\mid\beta$	$A\rightarrow \beta\{\ \varphi\ \}$
	$A\rightarrow(\varphi A)\mid\beta$	$A\rightarrow \{\ \varphi\ \}\beta$
3	$A\rightarrow \delta\ (\varphi \mid \lambda)\delta'$	$A\rightarrow \delta\ [\varphi\ ]\delta'$
4	$A\rightarrow[A]\varphi$	$A\rightarrow \{\ \varphi\ \}\varphi$

5	$A \rightarrow \beta$ β es simple o A ocurre pocas veces en la parte derecha de las producciones; y A no aparece en β.	Reemplace A por β siempre que aparezca en la parte derecha de alguna producción y elimine la producción $A \rightarrow \beta$.
6 Factorización	$A \rightarrow \delta (\varphi\beta_1 \mid \varphi\beta_2 \mid \ldots \mid \varphi\beta_n)\ \delta'$	$A \rightarrow \delta\ \varphi\ (\beta_1 \mid \beta_2 \mid \ldots \mid \beta_n)\ \delta'$
	$A \rightarrow \delta (\beta_1\ \varphi \mid \beta_2\varphi \mid \ldots \mid \beta_n\varphi)\ \delta'$	$A \rightarrow \delta (\beta_1 \mid \beta_2 \mid \ldots \mid \beta_n)\ \varphi\ \delta'$

Tabla 4.1 Reglas de Transformación de BNF a BNF extendido

4.2.2 Ejemplos

En esta sección mostramos ejemplos en los que transformamos gramáticas de la sección anterior al nuevo formalismo. Los ejemplos que se incluyen son: el de las expresiones aritméticas, el de gramáticas y el lenguaje para inventarios.

4.2.2.1 Expresiones aritméticas

Aplicamos las reglas a la gramática de la sección 4.1.3. Comenzamos una vez más con la regla 1.

1. E → E + T | E - T | T
2. T → T * F | T / F | F
3. F → - P | P
4. P → num | (E)

El | es asociativo entonces podemos suponer la siguiente asociación:

1. E → (E + T | E - T)| T
2. T → (T * F | T / F) | F
3. F → - P | P
4. P → num | (E)

En las producciones 1 y 2 podemos factorizar E y T a la izquierda respectivamente. En la producción 3 factorizar P a la derecha. Note que acá el símbolo lambda sí representa la cadena vacía.

1. E → E (+ T | - T)| T
2. T → T (* F | / F) | F
3. F → (-| λ) P

4. P → num | (E)

En las producciones 1 y 2 aplicar la ley de Arden (Regla 2) teniendo en cuenta que la recursión es por la izquierda.

1. E → T {(+ T | - T)}
2. T → F {(* F | / F)}
3. F → (-| λ) P
4. P → num | (E)

En la producción 3 aplicar la regla 3.

1. E → T {(+ T | - T)}
2. T → F {(* F | / F)}
3. F → [-] P
4. P → num | (E)

Podemos eliminar la regla 4, reemplazando P por su definición en la producción 3.

1. E → T {(+ T | - T)}
2. T → F {(* F | / F)}
3. F → [-](num | (E))

No vale la pena seguir factorizando para sacar la T en la producción 1 o F en la 2, ya que cuando pasemos a evaluar la semántica, es más fácil dejarlos como están.

4.2.2.2 El lenguaje de las gramáticas

La gramática para describir gramáticas independientes del contexto de la sección 4.1.4. Comenzamos aplicando la regla 1. Debemos enfatizar que el símbolo λ es un terminal y no la cadena vacía.

1. Gramatica → ({NTs}, {Ts}, noTerm, {Ps})
2. NTs → noTerm | (NTs , noTerm)
3. Ts → term | (Ts , term)
4. Ps → P | (Ps , P)
5. P → noTerm → ParteDerecha

6. ParteDerecha → λ | Sims
7. Sims → Sim | (Sims Sim)
8. Sim → term | noTerm

Ahora aplicamos la regla 2 a las producciones 2, 3, 4 y 7

1. Gramatica → ({NTs}, {Ts}, noTerm, {Ps})
2. NTs → noTerm { , noTerm }
3. Ts → term {, term }
4. Ps → P {, P }
5. P → noTerm → ParteDerecha
6. ParteDerecha → λ | Sims
7. Sims → Sim { Sim }
8. Sim → term | noTerm

Aplicamos la regla 5 a los terminales: NTs, Sims, Ts, Ps, y ParteDerecha .

1. Gramatica → ({ noTerm { , noTerm }}, { term {, term }}, noTerm, { P {, P }})
2. P → noTerm → (λ | (Sim { Sim }))
3. Sim → term | noTerm

4.2.2.3 El lenguaje de inventarios

Este lenguaje es relativamente sencillo de convertir. No hay muchos casos en que tengamos que aplicar la regla de Arden. Sin embargo, para facilitar la aplicación de diversas técnicas de compilación, se sugiere aplicar la regla de factorización para que dos opciones en el patrón de selección tengan el mismo prefijo.

Comenzamos, entonces, con la gramática obtenida en la sección anterior aplicando la primera regla de transformación.

1. COMs → COM EOL | COMs COM EOL
2. COM → INSERTAR_NUEVO nombre codigo precio cantidad
   | ACTUALIZAR CANTIDAD codigo num

| ACTUALIZAR CANTIDAD POR codigo num
| CAMBIAR PRECIO POR codigo porcentaje
| CAMBIAR PRECIO codigo precio
| PARA CADA CODIGO EN Codigos EOL ListaCs FINPARA

3. Codigos → OBR Cs CBR
4. Cs → codigo | Cs COMA codigo
5. ListaCs → COMi EOL | ListaCs COMi EOL
6. COMi →ACTUALIZAR CANTIDAD num
| ACTUALIZAR CANTIDAD POR num
| CAMBIAR PRECIO POR porcentaje
| CAMBIAR PRECIO precio

Aplicamos la ley de Arden a las producciones 1, 4 y 5.

1. COMs → COM EOL { COM EOL}
2. COM → INSERTAR_NUEVO nombre codigo precio cantidad
| ACTUALIZAR CANTIDAD codigo num
| ACTUALIZAR CANTIDAD POR codigo num
| CAMBIAR PRECIO codigo precio
| CAMBIAR PRECIO POR codigo porcentaje
| PARA CADA CODIGO EN OBR Codigos EOL CBR ListaCs FINPARA
3. Codigos → OBR Cs CBR
4. Cs → codigo { COMA codigo }
5. ListaCs → COMi EOL { COMi EOL}
6. COMi →ACTUALIZAR CANTIDAD num
| ACTUALIZAR CANTIDAD POR num
| CAMBIAR PRECIO POR porcentaje
| CAMBIAR PRECIO precio

Factorizamos ACTUALIZAR CANTIDAD y CAMBIAR PRECIO en las reglas 2 y 5. Y pasamos Cs a la producción 4 y eliminamos a 4.

1. COMs → COM EOL { COM EOL}
2. COM → INSERTAR_NUEVO nombre codigo precio cantidad
| ACTUALIZAR CANTIDAD (POR codigo num | codigo num)
| CAMBIAR PRECIO (POR codigo porcentaje | codigo precio)
| PARA CADA CODIGO EN Codigos EOL ListaCs FINPARA
3. Codigos → OBR codigo { COMA codigo } CBR
4. ListaCs → COMi EOL { COMi EOL}

5. COMi →ACTUALIZAR CANTIDAD (num | POR num)
   | CAMBIAR PRECIO (POR porcentaje | precio)

Se podría factorizar num a la derecha en las producciones dos y cinco. Sin embargo, así se simplificara la construcción de un compilador si-no, podría complicar su interpretación o traducción. Cuando pasemos al capítulo 5 veremos que hay una semántica distinta cuando tiene el POR que cuando no lo tiene.

4.3 Construyendo Compiladores

En el capítulo 3 vimos cómo dada cualquier gramática, G, podíamos construir un autómata no-determinístico que reconociera L(G). Se vieron dos tipos de autómatas: los descendentes y los ascendentes. A partir de una gramática en formato BNF extendido podemos construir un programa determinístico que reconoce el lenguaje generado por esa gramática siguiendo unas reglas sencillas de transformación. Para esto suponemos que ya tenemos un analizador léxico. Es decir, un programa que lee cadenas de caracteres y produce símbolos no terminales o tokens.

Vamos a suponer que tenemos los siguientes métodos que provee el analizador léxico:

- Token SigTok(): un método que retorna el siguiente token. Este sólo mira el token; no lo saca de la entrada.
- aceptar(Token t): este método saca el primer elemento de la cinta de entrada si es t. Si el siguiente símbolo no es t, genera un error.

Ahora describimos el problema:

Suponemos que para todo no terminal tenemos una sola regla de la forma:

$$A \rightarrow \beta_1\beta_2\ldots\beta_n$$

Donde cada βi puede ser de la forma:

- Un no-terminal.
- Un terminal.
- Una expresión de la forma $(\delta_1|\delta_2|\ldots|\delta_n)$.
- Una expresión de la forma $\{\delta\}$.
- Una expresión de la forma $[\delta]$.
- Lambda λ.

A su vez cada δ es de la forma: $\beta_1\beta_2\ldots\beta_n$

Para cada no terminal vamos a escribir un método: mA() que tendrá las instrucciones necesarias para reconocer cadenas generadas por A.

Si para el no terminal A tenemos la regla: $A \rightarrow \beta_1\beta_2\ldots\beta_n$, el método correspondiente sería de la forma:

```
mA()
{
   traducciónDe(β1)
   traducciónDe(β2)
   ....
   traducciónDe(βn)
}
```

Donde la traducción depende de la forma de (β_i) como se muestra en la Tabla 4.2. Se abusa de la notación usando el símbolo $\in$ y usando conjuntos, en lugar de simples valores, en los casos del switch y en las condiciones.

Tipo de Símbolo	**Traducción**
Terminal t	`accept(Token_t)`
noTerminal N	**`mN()`** donde `mN()` es el método que corresponde a N.

{δ}	`While sigToken()∈CS(δ) {`
	`  Traduccion(δ)`
	`}`

[δ]	`if sigToken()∈CS(δ) {`
	`  Traduccion(δ)`
	`}`

(δ_1|δ_2|… |δ_n)	`Token t;`
	`t = SigToken();`
	`Switch(t) {`
	`CS(δ1): Traduccion(δ1); break;`
	`CS(δ2): Traduccion(δ2); break;`
	…
	`CS(δn): Traduccion(δ2); break;`
	`Default:  throw new Exception();`
	`}`
	
	Nota: esto es una traducción correcta siempre y cuando, se cumpla la siguiente condición: $(\cap i \mid 1 \le i \le n: \delta_i) = \varnothing$.

Tabla 4.2 Generación de parser descendentes a partir de BNF extendido

Podemos generar un código un poco más eficiente aprovechando la forma de lo que se encuentra

dentro de los símbolos {} y [] como se muestra en la Tabla 4.3. Para esta tabla suponga que σ es un símbolo terminal.

Tipo de Símbolo	**Traducción**

{(σ_1|σ_2|… |σ_n) δ}	`While sigToken()∈{σ1,σ2,…,σn}) {`
	`  Avanzar();`
	`  Traduccion(δ);`
	`}`

[(σ_1|σ_2|… |σ_n) δ]	`if sigToken()∈ CS(δ) {`
	`  Avanzar();`
	`  Traduccion(δ)`
	`}`

(σ_1|σ_2|… |σ_n)	`if SigToken()∈{σ1,σ2,…,σn}`
	` avanzar()`
	`else`
	`throw new Exception();`
	`}`

(δ_1|$\sigma_i\delta_i$ |… |δ_n)	`Token t;`
	`t = SigToken();`
	`Switch(t) {`
	`CS(δ1): Traduccion(δ1); break;`
	`CS(δ2): Traduccion(δ2); break;`
	…
	`σi: avanzar();Traduccion(δi); break;`
	…
	`CS(δn): Traduccion(δ2); break;`
	`Default:  throw new Exception();`
	`}`

Tabla 4.3 Optimización de la traducción

En la Tabla 4.2 y en la Tabla 4.3, se usa el concepto de conjunto de selección (CS)[2]. El conjunto de selección de $\beta_1\beta_2\ldots\beta_n$ es el conjunto de terminales con los que puede comenzar $\beta_1\beta_2\ldots\beta_n$. Cada β_i puede ser un terminal, un no-terminal, o una expresión BNF de la forma: $\{\delta_1\delta_2\ldots\ \delta_k\}$, $[\delta_1\delta_2\ldots\ \delta_s]$, o $(\delta_1|\delta_2|\ldots\ |\delta_n)$.

Se presentan las siguientes reglas sencillas para calcular el conjunto de selección de $\beta_1\beta_2\ldots\beta_n$:

- Si n=0 (si no hay símbolos) entonces el conjunto de selección es vacío.

[2] Esta es una simplificación del cálculo del FIRST y el FOLLOW presentado en el libro de A. Aho, M., et al., Compilers – Principles, Techniques, & Tools, 2nd Ed., Addison Wesley, 2007.

- Si β_1 es un terminal (digamos t) entonces CS(t$\beta_2 \ldots \beta_n$)= {token_t} .
- Si β_1 es un no Terminal de A tal que su definición es: A→ $\varphi_1\varphi_2 \ldots \varphi_m$. entonces CS(A$\beta_2 \ldots \beta_n$)= CS($\varphi_1\varphi_2 \ldots \varphi_m\beta_2 \ldots \beta_n$).
- si β_1 es de la forma ($\delta_1|\delta_2| \ldots |\delta_m$):
  CS(($\delta_1|\delta_2| \ldots |\delta_m$) $\beta_2 \ldots \beta_n$) = CS($\delta_1\beta_2 \ldots \beta_n$) ∪CS($\delta_2\beta_2 \ldots \beta_n$) ∪ … ∪CS($\delta_m\beta_2 \ldots \beta_n$)
- si β_1 es de la forma {δ}:

  CS({δ} $\beta_2 \ldots \beta_n$) = CS(δ) ∪ CS($\beta_2 \ldots \beta_n$).

- si β_1 es de la forma [δ]:

  CS([δ] $\beta_2 \ldots \beta_n$) = CS(δ) ∪ CS($\beta_2 \ldots \beta_n$) .

Para el ejemplo de las expresiones aritméticas, ésta sería su traducción:

mExp() {	E →
mTerm();	T
While (sigToken()==”+” || sigToken()==”-”) {	{(+ T | - T)}
Token t = sigToken();	
switch (t) {	
case ‘+’:	
aceptar(‘+’);	
mTerm();	
Break;	
case ‘-’:	
aceptar(‘-’);	
mTerm();	
Break;	
Default:	
Throw new ParseException ()	
}	
}	
}	

mTerm() {	T →
mFact();	F
While (sigToken()==”*” || sigToken()==”/”) {	{(* F | / F)}
Token t = sigToken();	
switch (t) {	
case ‘*’:	
aceptar(‘*’);	
mFact();	
Break;	

case '/':	
aceptar('/');	
mFact();	
Break;	
Default:	
Throw new ParseException ()	
}	
}	
}	

mFact() {	F →
if (sigToken()=="-") {	[-]
Aceptar("-");	
}	
Token t = sigToken();	(num | (E))
switch (t) {	
case NUM:	
aceptar(NUM);	
Break;	
case '(':	
Aceptar('(');	
mExp();	
aceptar(')');	
Break;	
Default:	
Throw new ParseException ()	
}	
}	

Si aplicamos este proceso a la gramática extendida de las gramáticas, nos daría lo siguiente:

- **Gramatica → ({ noTerm { , noTerm }}, { term {, term }}, noTerm, { P {, P }})**
- **P → noTerm → (λ | (Sim { Sim }))**
- **Sim → term | noTerm**

Nos da los siguientes métodos:

mGramática()	
Aceptar("(");	**(**
Aceptar("{")	**{**
Aceptar(NoTerm);	**NoTerm**

While (sigToken()==",") {	**{**
Aceptar(",");	**,**
Aceptar(NoTerm);	**NoTerm**
}	**}**
Aceptar("}")	**}**
Aceptar(",")	**,**
Aceptar("{")	**{**
Aceptar("term")	**term**
While (sigToken()==","){	**{**
Aceptar(",");	**,**
Aceptar(term);	**term**
}	**}**
Aceptar("}")	**}**
Aceptar(",")	**,**
Aceptar(noTerm)	**noTerm**
Aceptar(",")	**,**
Aceptar("{")	**{**
mP()	**P**
While (sigToken()==","){	**{**
Aceptar(",");	**,**
mP()	**P }})**
}	**}**
Aceptar("}")	**}**
Aceptar(")")	**)**
}	

mP()	
Aceptar(noTerm);	**noTerm**
Aceptar("→")	**→**
Token t = sigToken() Switch (t) {	**(λ | (Sim { Sim }))**
Case "λ**":**	**λ**
Aceptar("λ");break;	
Case term**:**	
Case noterm:	**Sim**
pSim()	
While sigToken() in {term,noterm}{	**{**
pSim()	**Sim**
}	**}**
Break;	
}	
}	

msim()	
Token t = sigToken() Switch (t) {	**term** | **noTerm**
Case term:	**term**
Aceptar(term);break;	
Case noTerm:	**noTerm**
Aceptar(noTerm);break;	
}	
}	

4.4 JavaCC para generar Compiladores

En la sección anterior, mostramos cómo construir un compilador a partir de la definición en BNF de la gramática. Esto puede resultar bastante engorroso en lenguajes grandes. Existen muchos generadores automáticos de compiladores. En este libro usamos JavaCC, ub programa genera parsers de descenso recursivo. Lo bueno de esto es que el parser generado es fácil de entender. Otros enfoques generan parsers ascendentes, lo cual implica la necesidad de construir el autómata de pila. Por lo tanto el programa generado es la implementación del autómata de pila.

Tal como se mencionó en el capítulo 2, JavaCC permite definir tanto el analizador léxico como el sintáctico. Comenzamos, entonces definiendo los elementos sintácticos del lenguaje y luego mostramos cómo combinarlos para definir todas las cadenas del lenguaje.

Hay ciertas restricciones que tienen las gramáticas para las cuales podemos usar JavaCC. Como se usa un parser descendente, la cadena se lee de izquierda a derecha y la derivación que se usa es una derivación por la izquierda. La gramática no puede ser ambigua y no debe ser recursiva por la izquierda. JavaCC sí admite usar el formalismo extendido.

Un archivo para JavaCC tiene extensión .jj. Estos archivos se procesan con JavaCC, tal como se muestra en la Figura 1. En este ejemplo el archivo que se está procesando se llama Transform.jj. Al correr JavaCC, se generan los archivos: Transform.java, TransformTokenManager.java y TransformConstants.java, que son particulares al archivo procesado. También se generan los siguientes cuatro archivos que son comunes a todos: TokenMgrError.java, ParseException.java, Token.java, y ASCII_CharStream.java.

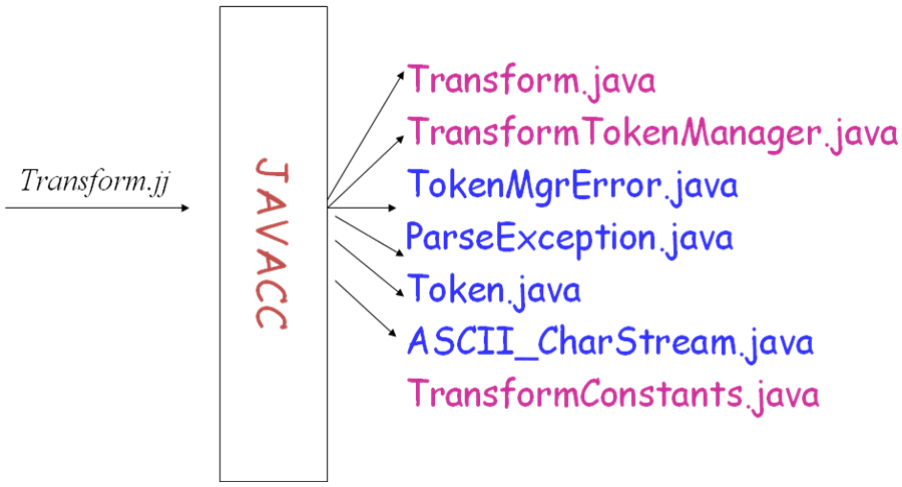

Figura 1. Procesamiento de Archivos JavaCC

Un archivo para JavaCC tiene el siguiente formato:

javacc_input →

[javacc_options]

PARSER_BEGIN "(" *IDENTIFIER* ")"

java_compilation_unit

PARSER_END "(" *IDENTIFIER* ")"

productions

<EOF>

IDENTIFIER debe ser el nombre del archivo sin la extensión jj. ***java_compilation_unit*** es simplemente código Java.

Las producciones se usan tanto para definir el analizador léxico como el sintáctico. Veremos cómo se usa JavaCC en algunos ejemplos.

Considere el lenguaje sencillo de expresiones aritméticas.

S → E \n

E → T { + T | - T }

T → F { * F | / F }

F → [-](num | (E))

Para definir el archivo JavaCC tendríamos primero que determinar cuáles son tokens del lenguaje. Si estos se componen de un solo carácter, no se tienen que definir explícitamente. Tenemos una sentencia para definir tokens. Es usual agrupar tokens que están relacionados. En este caso, el token DIGITO es privado al grupo. El token número es una secuencia de uno o más dígitos.

Tendríamos que crear un archivo que se llama ParseExpression.jj. La primera parte del archivo tendría lo siguiente. En este caso, no definimos nada adicional. Acá se pueden definir métodos privados o si se quiere hacer una aplicación sencilla se podría hacer todo en un solo archivo definiendo el main acá.

```
options
{
   LOOKAHEAD=1;
   IGNORE_CASE=true;
}

PARSER_BEGIN(ParseExpression)

  public class ParseExpression {
   }

PARSER_END(ParseExpression)
```

Los siguientes segmentos de código irían en la parte de producciones del archivo de JavaCC.

```
TOKEN :
{
    <#DIGITO: ["0" - "9"] >
|   <NUMERO: (<DIGITO>)+>
}
```

También debemos definir qué caracteres se ignoran y se usan como separadores, si existen. En este caso estos serían únicamente los espacios.

```
SKIP :
{
   "\r"
   | " "
}
```

Luego se definen las reglas. Como javaCC está escrito sobre Java y los corchetes se usan en Java para agrupar instrucciones, entonces para denotar la repetición de patronees se usa “(“ para abrir y “)*” para cerrar.

```
void input():
{}
{
  expr() "\n"
```

```
}
void expr():
{}
{
  term()( "+"  term() | "-" term())*
}
void term():
{}
{
  fact()( "*"  fact() | "/" fact())*
}

void fact():
{}
{
["-"] ( <NUMERO> |  "("  expr() ")")
}
```

Para invocar el parser, tendríamos que hacer lo siguiente:

```
ParseExpression parser = new ParseExpression(new java.io.StringReader(""));
String expression;
// Codigo para dejar en expresion lo que se quiere parsear

parser.ReInit(new java.io.StringReader(expression));

try {
          parser.input();
          System.out.println("Cadena Aceptada");
       }

         catch(ParseException pex)
         {     System.out.println("ERROR DE SINTAXIS");

         }
      catch (Error err)
      {
        System.out.println("Posible Error LExico");

      }
      catch (Exception error)
      {
        System.out.println("Otro Error");

      }
```

Considere el siguiente ejemplo de los inventarios. En este caso, debemos tener en cuenta la definición exacta de los elementos léxicos del lenguaje, los tokens.

Suponga que un código es un número de seis a nueve dígitos. Un precio es un número de punto flotante con dos cifras decimales precedido por el símbolo $. El porcentaje es un número de punto flotante con

dos cifras decimales, con el símbolo “%” al final y que puede estar precedido por el símbolo “-“. El nombre del producto se pone entre “<” y “>” y puede incluir blancos.

Los tokens para el analizador sintáctico de este archivo se describirían así:

```
SKIP :
{
"\r"
| " "
}
TOKEN :
{

    <INS_N: "INSERTAR_NUEVO" >
|   <ACT: "ACTUALIZAR" >
|   <CANT: "CANTIDAD" >
|   <CAMB: "CAMBIAR" >
|   <PRICE: "PRECIO" >
|   <POR: "POR" >
|   <PARA: "PARA" >
|   <CADA: "CADA" >
|   <CODIGO: "CODIGO" >
|   <EN: "EN" >
|   <FIN: "FINPARA">
}
TOKEN :
{
   <#LETRA: ["A" - "Z"] >
|   <#DIGITO: ["0" - "9"] >
|   <#PALABRA:  <LETRA>(<LETRA>|<DIGITO>)*>
|   <NOMBRE:  "<" (<PALABRA>|" ")* ">">
|   <NUMERO: (<DIGITO>)+>
|   <PRECIO: "$"(<DIGITO>)+"."<DIGITO><DIGITO>>
|   <PORCENTAJE: ("-")?(<DIGITO>)+"."(<DIGITO><DIGITO>)"%">
}
```

Para no confundir una <PALABRA> con las palabras reservadas, estas últimas deben definirse antes del token <PALABRA>.

Abajo se muestran las reglas. En algunas de ellas se hace un cálculo para obtener el valor semántico del token o para hacer validaciones adicionales. Para el nombre, se debe tomar la subcadena que está entre los corchetes triangulares. Para el precio y el porcentaje, se debe tomar el valor real de la cadena omitiendo el símbolo “$” y el símbolo “%”, respectivamente. Para el código, se debe verificar que tenga entre seis y nueve dígitos.

```
String nombre():
{
 Token tNom;
}
{
tNom=<NOMBRE>
{return  tNom.image.substring(1,tNom.image.length()-1); }
}

String codigo():
{
Token t;

}
{
 t = <NUMERO>
 {
 if (t.image.length() < 6 || t.image.length()> 9)
    throw new ParseException("Codigo invalido!");
 else
    return t.image;
 }
}

int cantidad():
{
Token t;
}
{
  t=<NUMERO>
{
    try {
            return (Integer.valueOf(t.image).intValue());
    } catch (NumberFormatException ee) {
    return 0;
 }
}
}

float precio():
{Token t;
 String cad;
 float value;
}
{
  t=<PRECIO>
{
      cad = t.image.substring(1);

  try {
      value = Float.valueOf(cad).floatValue();
  return value;
    } catch (NumberFormatException ee) {
    return 0;
    }}
}

float porcentaje():
{Token t;
 String cad;
 float value;
 int sign = 1;
}
{
  t=<PORCENTAJE>
{
      if (t.image.charAt(0)=='-') {
          cad = t.image.substring(1,t.image.length()-1);
          sign = -1;
      }
      else
      {
      cad = t.image.substring(0,t.image.length()-1);
      }
      try {
           value = Float.valueOf(cad).floatValue();
           return value*sign/100;
    } catch (NumberFormatException ee) {
    return 0;
    }}
}
```

Modificamos un poco la gramática para facilitar el proceso de interpretación que se discutirá en el siguiente capítulo. La gramática modificada se muestra a continuación:

1. COMs → (COM EOL)+
2. COM → COMIns | COMAct | COMCh | COMPara
3. COMIns → INSERTAR_NUEVO nombre codigo precio cantidad
4. COMAct → ACTUALIZAR CANTIDAD (POR codigo num | codigo num)
5. COMCh → CAMBIAR PRECIO (POR codigo porcentaje | codigo precio)
6. COMPara → PARA CADA CODIGO EN Codigos EOL ListaCs FINPARA
7. Codigos → OBR codigo { COMA codigo } CBR
8. ListaCs → (Inst EOL)+
9. Inst → InstAct | IntstCh
10. InstAct →ACTUALIZAR CANTIDAD (num | POR num)
11. InstCH → CAMBIAR PRECIO (POR porcentaje | precio)

Luego quitamos los terminales COM e Inst, y la gramática que traduciremos a JavaCC queda así:

1. COMs → ((COMIns | COMAct | COMCh | COMPara) EOL)+
2. COMIns → INSERTAR_NUEVO nombre codigo precio cantidad
3. COMAct → ACTUALIZAR CANTIDAD (POR codigo num | codigo num)
4. COMCh → CAMBIAR PRECIO (POR codigo porcentaje | codigo precio)
5. COMPara → PARA CADA CODIGO EN Codigos EOL ListaCs FINPARA
6. Codigos → OBR codigo { COMA codigo } CBR
7. ListaCs → ((InstAct | IntstCh) EOL)+
8. InstAct →ACTUALIZAR CANTIDAD (num | POR num)
9. InstCH → CAMBIAR PRECIO (POR porcentaje | precio)

Las demás reglas son una simple reescritura de la gramática. En este caso no se usan los valores semánticos de los terminales calculados arriba. Esto se verá en el capítulo de traducción del libro.

```
void input():
{}
{
((comandoInsert()|comandoAct() | comandoCh()| comandoPara())"\n")+
}


void comandoInsert():
{}
{
  <INS_N> nombre() codigo() precio() cantidad()
}

void comandoAct():
{ }
{
  <ACT> <CANT> (<POR>  codigo()  cantidad()  | codigo()  cantidad() )
}


void comandoCh():
{}
{
  <CAMB> <PRICE>( <POR>  codigo()  porcentaje()  | codigo()  precio())
}

void comandoPara():
{}
{
  <PARA> <CADA> <CODIGO> <EN> codigos() "\n"((instAct()| instCh())"\n")+ <FIN>
}

void codigos():
{}
{
  "{"  codigo()( "," codigo())*  "}"
}

void instAct():
{}
{
  <ACT> <CANT> (<POR>  cantidad()  | cantidad()  )
}


void instCh():
{}
{
  <CAMB> <PRICE>( <POR>  porcentaje() | precio())
}
```

Bibliografía:

1. Aho, A V., Sethi, R., Ullman, J.D., Compililadores: Principios, técnicas y herramientas, Addison-Wesley, 1990.
2. Manuales de JavaCC
3. López Rodrigo, Notas del curso de Compiladores, Universidad de los Andes, Bogotá, Colombia, 1986-1993.