Parsing Descendente

Hasta el momento hemos visto el proceso de compilación a grandes razgos, y hemos definido que la primera fase consiste en el análisis sintáctico. Esta fase a su vez la hemos divido en 2 procesos secuenciales: el análisis lexicográfico (*tokenización* o *lexing*), y el análisis sintáctico en sí (*parsing*). En esta sección nos concentraremos en este segundo proceso.

Recordemos que el proceso de parsing consiste en analizar una secuencia de tokens, y producir un árbol de derivación, lo que es equivalente a producir una derivación extrema izquierda o derecha de la cadena a reconocer. Tomemos la siguiente gramática, que representa expresiones aritméticas (sin ambigüedad):

```
E = T
  | T + E

T = int * T
  | int
  | (E)
```

Tomemos la cadena siguiente:

```
2 * (3 + 5)
```

Que una vez procesada por la fase lexicográfica produce la siguiente secuencia de *tokens*.

```
int  *  (  int  +  int  )
```

Intentemos producir una derivación (en principio, extrema izquierda, por comodidad) de esta cadena. Como sabemos, si existe una derivación extrema izquierda, es porque se cumple que:

```
[1]  E -*-> int * ( int + int )
```

Preguntémonos entonces, ¿de cuántas formas pudiera E derivar en la cadena? Evidentemente, hay exactamente dos formas en que E es capaz de producir esta cadena, es decir, hay solamente dos producciones posibles que pudieran seguir en la derivación extrema izquierda. O bien `E -> T + E` o bien `E -> T`. Probemos entonces con la primera de ellas:

```
[2]  E -*-> T -*-> int * ( int + int )
```

Como estamos produciendo una derivación extrema izquierda, tenemos que expandir a continuación el símbolo T. Nuevamente, hay varias opciones, probemos con la primera:

```
[3]  E -*-> int * T -*-> int * ( int + int )
```

De momento parece que vamos por buen camino, pues hemos logrado producir los primeros 2 *tokens* de la cadena. Expandimos entonces nuevamente el no-terminal más a la izquierda con la primera producción posible T -> int * T:

```
[4]  E -*-> int * int * T -*-> int * ( int + int )
```

En este punto, podemos darnos cuenta de que hemos tomado el camino equivocado. Cómo estamos produciendo una derivación extrema izquierda, y los terminales no derivan en ningún símbolo, sabemos que todo lo que esté a la izquierda del primer no-terminal no va a cambiar en el futuro. Luego, es evidentemente que ya no seremos capaces de generar la cadena, pues estos terminales (int * int *) no son prefijo de la cadena a reconocer. Deshagamos entonces la última producción (volviendo al paso 3) y probemos otro camino (T -> int):

```
[4]  E -> int * int -*-> int * ( int + int )
```

Nuevamente, hemos generado una secuencia de *tokens* que no es prefijo de la cadena. Probamos de nuevo:

```
[4]  E -*-> int * ( E ) -*-> int * ( int + int )
```

Parece que en este punto hemos hecho un avance, pues logramos reconocer tres *tokens* de prefijo. Expandimos entonces el nodo E -> T nuevamente, y de ahí probamos la próxima producción (T -> int * T), que también falla:

```
[5]  E -*-> int * ( T ) -*-> int * ( int + int )
[6]  E -*-> int * ( int * T ) -*-> int * ( int + int )
```

Probando con cualquiera de las producciones de T nunca podremos generar el *token* + que falta, por lo tanto eventualmente volveremos a probar con E -> T + E:

```
[5]  E -*-> int * ( T + E ) -*-> int * ( int + int )
```

Una vez llegados a este punto, ya podemos hacernos una idea de cómo funciona este proceso. Eventualmente, tendremos que derivar T -> int y E -> T -> int para lograr producir la cadena final. Finalmente obtenemos la derivación extrema izquierda siguiente:

```
E -> T
  -> int * T
  -> int * ( E )
  -> int * ( T + E )
  -> int * ( int + E )
  -> int * ( int + T )
  -> int * ( int + int )
```

Básicamente, lo que hemos hecho ha sido probar todas las posibles derivaciones extrema izquierda, de forma recursiva, podando inteligentemente cada vez que era evidente que habíamos producido una derivación

incorrecta. Tratemos de formalizar este proceso.

Parsing Recursivo Descendente

De forma general, tenemos tres tipos de operaciones o situaciones que analizar:

- La expansión de un no-terminal en la forma oracional actual.
- La prueba recursiva de cada una de las producciones de este no-terminal.
- La comprobación de que un terminal generado coincide con el terminal esperado en la cadena.

Para cada una de estas situaciones, vamos a tener un conjunto de métodos. Para representar la cadena a reconocer, necesitamos mantener un estado global que indique la parte reconocida de la cadena. Diseñemos una clase para ello:

```
interface IParser {
    bool Parse(Token[] tokens);
}

class RecursiveParser : IParser {
    Token[] tokens;
    int nextToken;

    //...
}
```

Para reconocer un terminal, tendremos un método cuya función es a la vez decidir si se reconocer el terminal, y avanzar en la cadena:

```
bool Match(Token token) {
    return tokens[nextToken++] == token;
}
```

A cada no-terminal vamos a asociar un método recursivo cuya función es determinar si el no-terminal correspondiente genera una sub-cadena "adecuada" del lenguaje. Por ejemplo, para el caso de la gramática anterior, tenemos los siguientes métodos:

```
bool E() {
    // Parsea un no-terminal E
}

bool T() {
    // Parsea un no-terminal T
}
```

La semántica de cada uno de estos métodos es que devuelven true si y solo si el no-terminal correspondiente genera una parte de la cadena, comenzando en la posición nextToken. Tratemos de escribir el código del método

E. Para ello, recordemos que el símbolo E deriva en 2 producciones: E -> T y E -> T + E. Por tanto, de forma recursiva podemos decir que E genera esta cadena si y solo si la genera a partir de una de estas dos producciones. El primer caso es fácil: E genera la cadena a partir de derivar en T si y solo si T a su vez genera dicha cadena, y ya tenemos un método para eso:

```
bool E1() {
    // E -> T
    return T();
}
```

Para el segundo caso, notemos que la producción E -> T + E básicamente lo que dice es: necesitamos generar una cadena a partir de E, de forma tal que primero se genere una parte con T, luego se genere un + y luego se genere otra parte con E. Dado que ya tenemos todos los métodos necesarios:

```
bool E2() {
    // E -> T + E
    return T() && Match(Token.Plus) && E();
}
```

Aprovechamos el operador && con cortocircuito para podar lo antes posible el intento de generar la cadena, de forma que el primero de estos tres métodos que falle ya nos permite salir de esa rama recursiva. Ahora que tenemos métodos para cada producción, podemos finalmente develar el cuerpo del método E:

```
bool E() {
    int currToken = nextToken;
    if (E1()) return true;

    nextToken = currToken;
    if (E2()) return true;

    return false;
}
```

Este método simplemente prueba cada una de las producciones en orden, teniendo cuidado de retornar nextToken a su valor original tras cada llamado recursivo fallido. Así mismo, podemos escribir el método asociado al símbolo T, basado en los métodos correspondientes a cada producción:

```
bool T1() {
    // T -> int * T
    return Match(Token.Int) && Match(Token.Times) && T();
}

bool T2() {
    // T -> int
    return Match(Token.Int);
```

```
}

bool T3() {
    // T -> (E)
    return Match(Token.Open) && E() && Match(Token.Closed)
}
```

Y el método general para T queda así:

```
bool T() {
    int currToken = nextToken;
    if (T1()) return true;

    nextToken = currToken;
    if (T2()) return true;

    nextToken = currToken;
    if (T3()) return true;

    return false;
}
```

Es posible hacer estos métodos más compactos introduciendo un nuevo método auxiliar:

```
bool Reset(int pos) {
    nextToken = pos;
    return true;
}
```

Este método nos permite reescribir cada método con una sola expresión, haciendo uso del operador || con cortocircuito:

```
bool E() {
    int n = nextToken;
    return E1() || Reset(n) && E2();
}

bool T() {
    int n = nextToken;
    return T1() || Reset(n) && T2() || Reset(n) && T3();
}
```

Finalmente, para reconocer la cadena completa, solo nos queda garantizar que se hayan consumido todos los *tokens*:

```
bool Parse(Token[] tokens) {
    return E() && nextToken == tokens.Length;
}
```

Esta metodología para crear parsers recursivos descendentes puede ser aplicada fácilmente a cualquier gramática libre del contexto. Sin embargo, no todas las gramáticas pueden ser reconocidas de esta forma. Según la estructura de la gramática, es posible que es parser definido no funcione correctamente.

Por ejemplo, para gramáticas ambigüas, el parser (si termina) dará alguna de las derivaciones extrema izquierda posibles, en función del orden en que hayan sido definidas las producciones. Esto se debe al uso de operadores con cortocircuito. Es posible modificar este tipo de parsers fácilmente para generar no el primero sino todas las derivaciones extrema izquierda disponibles, simplemente reemplazando los operadores || más externos.

Consideremos ahora la siguiente gramática:

```
S -> Sa | b
```

Para esta gramática, el parser recursivo descendente queda de la siguiente forma (simplificada):

```
bool S() {
    int c = nextToken;
    return S() && Match(Token.A) || Reset(n) && Match(Token.B);
}
```

El problema evidente con este parser es que al intentar reconocer el símbolo S el algoritmo cae en una recursión infinita. Este tipo de gramáticas se denominan gramáticas con recursión izquierda, que definiremos así:

> **Definición**: Una gramática libre del contexto G=<S,N,T,P> se dice recursiva izquierda si y solo si S -*-> Sw (donde w es una forma oracional).

La forma más sencilla de las gramáticas recursivas izquierdas es cuando existe directamente una producción S -> Sw. A este caso le llamamos *recursión izquierda directa*. Para este caso, es posible eliminar la recursión izquierda de forma sencilla. Tomemos nuevamente la gramática anterior:

```
S -> Sa | b
```

Es fácil ver que esta gramática general el lenguaje ba*. Otra gramática que genera dicho lenguaje sin recursión izquierda es:

```
S -> bX
X -> aX | epsilon
```

Aún cuando a y b son formas oracionales en general, y no simplemente terminales, el patrón anterior es válido. De forma general, si una gramática tiene recursión izquierda de la forma:

```
S -> Sa1 | Sa2 | ... San | b1 | b2 | ... | bm
```

Es posible eliminar la recursión izquierda con la transformación:

```
S -> b1X | b2X | ... | bmX
X -> a1X | a2X | ... | anX | epsilon
```

Para el caso más general de recursión izquierda indirecta, también existe un algoritmo para su eliminación, pero de momento no lo presentaremos 😦

El algoritmo de parsing que hemos desarrollado resuelve, al menos de forma teórica, el problema de construir el árbol de derivación. Aunque el código presentado no construye explícitamente el árbol de derivación, es bastante fácil modificarlo al respecto. Sin embargo, aunque en principio el problema ha sido resuelto, el algoritmo recursivo descendente es extremadamente ineficiente. El problema es que, en principio, es necesario probar con todos los árboles de derivación posibles antes de encontrar el árbol correcto. De cierta forma, para resolver el problema de parsing lo que hemos hecho es buscar entre todos los posibles programas, cuál de ellos tiene una representación textual igual a la cadena deseada.

Parsing Predictivo Descendente

Idealmente, quisiéramos un algoritmo de parsing que construya el árbol de derivación con un costo lineal con respecto a la cadena de entrada. Para ello, necesitamos poder "adivinar", en cada método recursivo, cuál es la rama adecuada a la que descender. Con vistas a resolver este problema, consideremos nuevamente la gramática vista en la sección anterior:

```
E -> T | T + E
T -> int * T | int | (E)
```

Analicemos ahora nuevamente la cadena de *tokens* int * (int + int), y tratemos de "adivinar" en cada paso de una derivación extrema izquierda qué producción es necesario aplicar. Tengamos en cuenta que en cada paso del proceso de parsing, hay al menos un *token* que conocemos tiene que ser generado de inmediato (dado que la gramática no puede "intercambiar" *tokens* una vez generados). Por tanto, observando el siguiente *token* que es necesario generar (nextToken en nuestra implementación), tenemos una pista de cuáles producciones no son posibles. En el caso anterior, el primer *token* a generar es int. Por tanto, es evidente que ninguna producción que derive en (E) funciona, pues el (no coincidirá con el *token* int. La primera producción a aplicar tiene que derivar en una forma oracional que comience por int.

Desafortunadamente existen varias producciones que generan un int al inicio. Tanto T -> int como T -> int * T pudieran ser escogidas. Aún más, a cualquiera de estas dos producciones se llega tanto por E -> T como por E -> T + E. Por tanto, hay varios caminos por los cuáles se pudiera generar el primer *token* int. Intuitivamente, esto se debe a que la gramática no está **factorizada**. Informalmente, llamaremos a una gramática **factorizada a la izquierda** si las producciones de cualquier símbolo, dos a dos, no comparten ningún prefijo.

En muchas ocasiones es fácil factorizar una gramática. Se introduce un no-terminal nuevo por cada grupo de producciones que compartan un prefijo común. Se reemplazan dichas producciones por una sola producción

nueva que contiene el prefijo común, y el nuevo no-terminal se hace derivar en todos los posibles sufijos:

```
E -> T X
X -> + E | epsilon
T -> int Y | (E)
Y -> * T | epsilon
```

Por supuesto, es posible que la relación entre los prefijos sea más complicada, y una vez que se realice la transformación anterior aún queden producciones no factorizadas (e.j. X -> abC | abD | aY). Incluso en estos casos es posible factorizar la gramática aplicando varias veces el proceso de factorización anterior.

Esta modificación evidentemente no cambia el lenguaje, y ni siquiera cambia la ambigüedad o no de la gramática. Simplemente nos permite delegar la decisión de que producción tomar un *token* hacia adelante. Si antes no sabíamos cuando venía int que producción tomar, porque podía ser T -> int o T -> int * T, ahora simplemente reconocemos el primer int, y delegamos la decisión de generar epsilon o * T a un nuevo no-terminal.

> **Nota:** Desde el punto de vista del diseño de lenguajes, el beneficio de este cambio es discutible. Por un lado nos permite aplicar un algoritmo que de otra forma no funcionaría. Sin embargo, por otra parte, estamos provocando un cambio en el diseño de la gramática, que es una cuestión de "alto nivel", para poder usar un algoritmo particular, que es una cuestión de "bajo nivel". En otras palabras, estamos cambiando el diseño en función de la implementación. Este cambio puede tener efectos adversos. Por ejemplo, nuestra gramática para expresiones aritméticas es ahora más difícil de entender, pues contiene símbolos "extraños" que no significan nada desde el punto de vista semántico, solamente están ahí para simplificar la implementación. El árbol de derivación ahora es más complejo. Más adelante discutiremos esta problemática en mayor profundidad.

Consideremos ahora nuevamente el proceso de parsing, y tratemos de ver si es posible adivinar en todo caso cuál producción aplicar. Tomemos como ejemplo la cadena int * (int + int), y tratemos de generar la derivación extrema izquierda:

```
E -*-> int * ( int + int )
```

La primera producción tiene que ser necesariamente E -> T X pues es la única disponible:

```
E -*-> T X -*-> int * ( int + int )
```

Ahora, tenemos que expandir el primer símbolo T. Afortunadamente, sabemos que esta expansión obligatoriamente genera un *token* a continuación, ya sea int o (. Por tanto es trivial escoger la única producción posible:

```
E -*-> int Y X -*-> int * ( int + int )
```

El nuevo símbolo a expandir es Y, y ahora se nos complica un poco el análisis. Y bien pudiera desaparecer (Y -> epsilon) o generar un * T. Sabemos que la producción * T nos genera el token que queremos. Pero la pregunta es, ¿cómo sabemos que la producción Y -> epsilon no pudiera redundar en que eventualmente aparezca ese *

por otro lado? Observando la gramática, intuitivamente, podemos ver que si Y desaparece, X nunca podrá poner un * justamente en esa posición, ya que X tiene que generar primero un + (X -> + E) antes de que aparezca otro no-terminal que pudiera generar el *. Por tanto, la única producción posible es Y -> * T:

```
E -*-> int * T X -*-> int * ( int + int )
```

En este punto es fácil ver que la única solución es derivar T -> (E), pues T nunca desaparece:

```
E -*-> int * ( E ) X -*-> int * ( int + int )
```

Ahora volvemos al punto inicial:

```
E -*-> int * ( T X ) X -*-> int * ( int + int )
```

Nuevamente T tiene que generar un int:

```
E -*-> int * ( int Y X ) X -*-> int * ( int + int )
```

Evidentemente Y no puede generar el token + que hace falta, así que solo puede desaparecer:

```
E -*-> int * ( int X ) X -*-> int * ( int + int )
```

Volvemos entonces a la situación complicada anterior. Es cierto que X -> + E nos sirve, pero ¿cómo sabemos que es la única opción? ¿Es posible que de X -> epsilon se logre en algún momento que aparezca un +. Si miramos la forma oracional generada hasta el momento, vemos que no nos queda otra X dentro los paréntesis que pueda poner el + que falta. Sin embargo, este análisis no lo puede hacer nuestro algoritmo, que solamente conoce el no-terminal actual, y el siguiente token que es necesario generar. Tratemos de hacer un razonamiento un más generalizable. La pregunta que estamos haciendo aquí básicamente es si es conveniente eliminar X con la esperanza de que aparezca un + de lo que sea que venga detrás. Más adelante formalizaremos este concepto, pero por ahora baste decir que, intuitivamente, podemos ver que detrás de una X solamente puede venir o bien un) o bien el fin de la cadena. Por tanto, no queda otra opción que derivar X -> + E:

```
E -*-> int * ( int + E ) X -*-> int * ( int + int )
```

Generar el siguiente int es fácil:

```
E -*-> int * ( int + T X ) X -*-> int * ( int + int )
E -*-> int * ( int + int Y X ) X -*-> int * ( int + int )
```

Finalmente nos queda por generar un). Claro que si miramos la forma oracional generada, sabemos que es necesario eliminar Y y X, pero recordemos que nuestro algoritmo no puede ver tan hacia adelante. De todas formas, no hace falta mirar más, ni Y ni X son capaces de generar nunca un), así que ambos símbolos desaparecen:

```
E -*-> int * ( int + int X ) X -*-> int * ( int + int )
E -*-> int * ( int + int ) X -*-> int * ( int + int )
```

Y finalmente la última X debe desaparecer también pues se ha generado toda la cadena. Finalmente nos queda:

```
E -> T X
  -> int Y X
  -> int * T X
  -> int * ( E ) X
  -> int * ( T X ) X
  -> int * ( int Y X ) X
  -> int * ( int X ) X
  -> int * ( int + E ) X
  -> int * ( int + T X ) X
  -> int * ( int + int Y X ) X
  -> int * ( int + int X ) X
  -> int * ( int + int ) X
  -> int * ( int + int )
```

La derivación extrema izquierda producida es considerablemente mayor con esta gramática factorizada, dado que existen más producciones. Sin embargo, ganamos en un factor exponencial al eliminar el *backtrack* por completo. Intuitivamente, el largo de esta derivación debe ser lineal con respecto a la longitud de la cadena de entrada, pues a lo sumo en cada paso o bien generamos un nuevo *token* o derivamos el símbolo más izquierdo en nuevos símbolos. Dado que no tenemos recursión izquierda, estas operaciones con cada símbolo no pueden ser "recursivas". Es decir, si el no-terminal más izquierdo es Y, y empezamos a derivarlo, eventualmente terminaremos con ese Y, ya sea produciendo un terminal o derivando en epsilon. De forma general, el costo está acotado superiormente por |w| * |N|, pues no es posible que para generar un *token* sea necesario usar |N| + 1 no terminales, ya que en ese caso tendría un no-terminal derivando en sí mismo (al menos de forma indirecta), lo que contradice que la gramática no tenga recursión izquierda.

Tratemos ahora de formalizar este proceso de "adivinación" de qué producción aplicar en cada caso. De forma general nos hemos enfrentado a dos interrogantes fundamentalmente distintas:

- Saber si alguna de las producciones de X puede derivar en una forma oracional cuyo primer símbolo sea el terminal que toca generar.
- Si X -> epsilon, saber si esta derivación puede potencialmente redundar en que "lo que sea que venga detrás" de X genere el terminal que toca.

Si para las preguntas anteriores obtenemos una sola producción como respuesta, entonces podemos estar seguros de la decisión a tomar. En caso de que obtengamos más de una respuesta, nuestro algoritmo no podrá decidir qué producción tomar, y será inevitable el *backtrack*. Para encontrar una respuesta a estas preguntas, intentemos formalizar estos conceptos de "X puede derivar en..." y "lo que venga detrás de X...".

Llamaremos First(W) al conjunto de todos los terminales que pueden ser generados por W como primer elemento (siendo W una forma oracional cualquiera, no solamente un no-terminal). Formalmente:

> **Definición**: Sea G=<S,N,T,P> una gramática libre del contexto, W \in { N \union T }* una forma oracional, y x \in T un terminal. Decimos que x \in First(W) si y solo si W -*-> xZ (donde W \in { N \union T }* es otra forma oracional).

Este concepto captura formalmente la noción de "comenzar por". De forma intuitiva, si logramos computar el conjunto First(W) para todas las producciones X -> W de nuestra gramática, y cada uno de estos conjuntos de las producciones del mismo símbolo son disjuntos dos a dos, entonces podremos decir inequívocamente qué producción aplicar para generar el terminal que toca (o cuando no es posible generarlo). Notemos que fue necesario definir First(W) no solo para un no-terminal, sino para una forma oracional en general, pues necesitamos computarlo en toda parte derecha de una producción.

Por otro lado, la noción de "lo que viene detrás" se formaliza en un concepto similar, denominado Follow(X). En este caso solo necesitamos definirlo para un no-terminal, pues solo nos interesan las producciones X -> epsilon. Informalmente diremos que el Follow(X) son todos aquellos terminales que pueden aparecer en cualquier forma oracional, detrás de un no-terminal X. Formalmente:

> **Definición**: Sea G=<S,N,T,P> una gramática libre del contexto, X \in N un no-terminal, y x \in T un terminal. Decimos que x \in Follow(X) si y solo si S -*-> WXxZ (donde W, Z \in { N \union T }* son formas oracionales cualesquiera).

La definición de Follow(X) básicamente nos dice que si en algún momento el terminal que queremos generar x está justo detrás del no-terminal X que toca expandir, entonces X -> epsilon es una producción válida a aplicar, porque existe la posibilidad de que otro no-terminal genere a x justo en esa posición (aunque en la cadena particular que se está reconociendo puede que esto no sea posible).

Supongamos entonces que tenemos todos estos conjuntos calculados (o potencialmente calculables en cualquier momento). ¿Cómo podemos utilizarlos para guiar la búsqueda del árbol de derivación correcto? Veamos como podría quedar el método recursivo descendente para generar T en esta nueva gramática:

```
bool T() {
    // T -> int Y
    if (Tokens[currToken] == Token.Int)
        return Match(Token.Int) && Y()

    // T -> ( E )
    else if (Tokens[currToken] == Token.Open)
        return Match(Token.Open) && E() && Match(Token.Close);

    return false;
}
```

Como T siempre genera un token, es fácil decidir qué camino escoger. Por otro lado, en la expansión de X, es posible que sea necesario escoger X -> epsilon. En este caso, el método recursivo sería:

```
bool X() {
    // X -> + E
    if (Tokens[nextToken] == Token.Plus)
```

```
        return Match(Token.Plus) && E()

    // X -> epsilon
    else if (Follow("X").Contains(Tokens[currToken]))
        return true;

    return false;
}
```

En el caso de X -> epsilon, simplemente retornamos true de inmediato y no consumimos el terminal correspondiente.

De forma general, podemos escribir cualquier método recursivo descendente de la siguiente forma (asumimos algunos métodos y clases utilitarios que no presentaremos formalmente):

```
bool Expand(NonTerminal N) {
    foreach(var p in N.Productions) {
        if (!p.IsEpsilon && First(p).Contains(Tokens[nextToken]))
            return MatchProduction(p);

        if (p.IsEpsilon && Follow(N).Contains(Tokens[nextToken]))
            return true;
    }

    return false;
}
```

El método MatchProduction puede a grandes razgos implementarse de la siguiente forma:

```
bool MatchProduction(Production p) {
    foreach(var symbol in p.Symbols) {
        if (symbol.IsTerminal && !Match(symbol as Token))
            return false;

        if (!symbol.IsTerminal && !Expand(symbol as NonTerminal))
            return false;
    }

    return true;
}
```

En todos estos casos hemos asumido que la primera producción aplicable era la única posible. Para ello deben cumplirse ciertas restricciones entre los conjuntos First y Follow que formalizaremos a continuación.

Gramáticas LL(1)

Llamaremos gramáticas LL(1) justamente a aquellas gramáticas para las cuales el proceso de cómputo de First y Follow descrito informalmente en la sección anterior nos permite construir un parser que nunca tenga que hacer *backtrack*. El nombre LL(1) significa *left-to-right left-derivation look-ahead 1*. Es decir, la cadena se analiza de izquierda a derecha, se construye una derivación extrema izquierda, y se analiza un solo *token* para decidir que producción aplicar. De forma general, existen las gramáticas LL(k), donde son necesarios k *tokens* para poder predecir que producción aplicar. Aunque los principios son los mismos, el proceso de construcción de estos conjuntos es más complejo, y por lo tanto no analizaremos estas gramáticas por el momento 😦

Para poder formalizar este concepto, será conveniente primero encontrar algoritmos explícitos para computar los conjuntos First y Follow. Comencemos por el First 😉. Veamos primero algunos hechos interesantes que se cumplen en este conjunto, y luego veremos cómo se diseña un algoritmo para su cómputo. No presentaremos demostración para estos hechos, pues la mayoría son intuitivos.

- Si X -> W1 | W2 | ... | Wn entonces por definición, First(X) = \union First(W_i).
- Si X -*-> epsilon entonces epsilon \in First(X).
- Si W = xZ donde x es un terminal, entonces trivialmente First(W) = { x }.
- Si W = YZ donde ambos Y y Z son no-terminales, entonces First(Y) \subseteq First(W).
- Si W = YZ y Y -*-> epsilon entonces First(Z) \subseteq First(W).

Las observaciones anteriores nos permiten diseñar un algoritmo para calcular todos los conjuntos First(X) para cada no-terminal X. Como de forma general pueden existir producciones recursivas, calcularemos todos los conjuntos First a la vez, aplicando cada una de las "reglas" anteriores, hasta que no se modifique ninguno de los conjuntos First. Nuevamente abusaremos de la imaginación y creatividad para introducir métodos y clases utilitarias sin definirlos de manera formal.

```
Firsts CalculateFirsts(Grammar G) {
    var Firsts = new Firsts(); // Parecido a un Diccionario

    // Calculamos el First de cada terminal
    foreach(var t in G.Terminals) {
        Firsts[t] = new FirstSet() { t };
    }
    foreach(var T in G.NonTerminals) {
        Firsts[T] = new FirstSet(); // Parecido a un HashSet
    }

    bool changed = true;

    do {
        changed = false;

        // Vamos por cada producción
        foreach(var p in G.Productions) {
            // X -> W
            var X = p.Left;
            var W = p.Right;
```

```
            if (p.IsEpsilon) { // X -> epsilon
                changed = Firsts[X].Add(epsilon);
            }
            else {
                foreach(var s in W) {
                    bool allEpsilon = true;

                    // Agregamos todo en el First(s)
                    changed = Firsts[X].AddAll(Firsts[s]);

                    // Si s_i deriva en epsilon,
                    // agregamos también el First(s_i+1)
                    if (!Firsts[s].Contains(epsilon)) {
                        allEpsilon = false;
                        break;
                    }
                }

                // Si todos los s_i derivan en epsilon
                // entonces epsilon pertenece al First(X)
                if (allEpsilon) {
                    changed = Firsts[X].Add(epsilon);
                }
            }
        }
    } while (changed);

    return Firsts;
}
```