4. Herramientas para el análisis sintáctico

- Definición y conceptos
- Gramáticas libres de contexto
- Arboles de sintaxis
- Parser recursivo descendente
- Patrones

Definición y conceptos

- Mientras que el análisis léxico divide la entrada en *tokens*, el propósito del análisis sintáctico (conocido como *parsing*) es recambiar estos *tokens*. No de regreso a una lista de caracteres, sino a una estructura de datos (árbol de sintaxis) que refleje la estructura del texto.

- Además de encontrar la estructura del texto de entrada, el análisis sintáctico debe rechazar texto inválido reportando errores.

- Para el análisis sintáctico se utilizan métodos más avanzados, sin embargo se sigue la misma estrategia: Una notación entendible por el humano se transforma en una notación máquina de bajo nivel que servirá para una ejecución eficiente. Este proceso es llamado generación de *parser*.

- La notación que utilizamos para los humanos son gramáticas libres de contexto.

Gramáticas libres de contexto

- Al igual que las expresiones regulares, las gramáticas libres de contexto (GLC) describen conjuntos de cadenas, por ejemplo lenguajes.
- Además, una gramática libre de contexto también define estructuras sobre las cadenas del lenguaje que define.
- Un lenguaje se define sobre un alfabeto, por ejemplo el conjunto de tokens producidos por un lexer o el conjunto de caracteres alfanuméricos.
- Los símbolos en el alfabeto se llaman ***terminales***.
- Una GLC define de manera recursiva varios conjuntos de cadenas. Cada conjunto se denota por un nombre, el cual es llamado ***no terminal***.
- Uno de los no terminales es designado como el ***símbolo inicial*** de la gramática.

Gramáticas libres de contexto

- Los conjuntos son descritos por producciones.
- Cada producción describe algunas de las posibles cadenas contenidas en el conjunto de no terminales.
- La forma de una producción es:
    - $N \longrightarrow X_1, X_2, \ldots X_n$.
- N es un no terminal y $X_1 \ldots X_n$ Son cero o más símbolos terminales o no terminales.
- El significado de la notación es que el conjunto denotado por N contiene cadenas que se obtienen al concatenar $X_1 \ldots X_n$.

Gramáticas libres de contexto - Ejemplos

- **A —> a**
  - El conjunto denotado por el no terminal **A** contiene la cadena de un carácter **a**.
- **A —> aA**
  - El conjunto denotado por **A** contiene todas las cadenas formadas por una **a** seguida de la cadena tomada del conjunto denotado por **A**.
  - Si juntamos las dos producciones, estas indican que **A** contiene todas las secuencias no vacías de **a** y es equivalente a la expresión regular **a+**.

Gramáticas libres de contexto - Ejemplos

- **B —>**
- **B —> aB**
  - Estas producciones definen una gramática equivalente a la expresión regular **a***.
  - La primera producción indica que la cadena vacía es parte del conjunto **B**.
  - Las producciones que tienen vacía la parte derecha, se llaman **producciones vacías**. También se puede utilizar **ε**, en la parte derecha.

Gramáticas libres de contexto - Ejemplos

- El lenguaje de las cadenas $\mathbf{a^n b^n}$, se puede describir con la gramática:
  - S —>
  - S —> aSb

Gramáticas libres de contexto - Ejemplos

- T —> R
- T —> aTa
- R —> b
- R —> bR
    - Esta gramática tiene como símbolo inicial **T** y denota el conjunto de cadenas que inician con cualquier número de **a** seguido de una o más **b** y después la misma cantidad de **a** con la que se inició.
    - La gramática puede ser escrita de manera más compacta:
    - T—> R | aTa
    - R —> b | bR

Gramáticas libres de contexto - Equivalencias con expresiones regulares

Form of s_i	Productions for N_i
ε	$N_i \rightarrow$
$\mathtt{a}$	$N_i \rightarrow \mathtt{a}$
$s_j s_k$	$N_i \rightarrow N_j N_k$
$s_j \vert s_k$	$N_i \rightarrow N_j$ $N_i \rightarrow N_k$
s_j*	$N_i \rightarrow N_j N_i$ $N_i \rightarrow$
s_j+	$N_i \rightarrow N_j N_i$ $N_i \rightarrow N_j$
$s_j?$	$N_i \rightarrow N_j$ $N_i \rightarrow$

Gramáticas libres de contexto - Más ejemplos

- Dos gramáticas simples: una para expresiones y otra para enunciados.

$$
\begin{aligned}
Exp &\rightarrow Exp + Exp \\
Exp &\rightarrow Exp - Exp \\
Exp &\rightarrow Exp * Exp \\
Exp &\rightarrow Exp / Exp \\
Exp &\rightarrow \textbf{num} \\
Exp &\rightarrow (\ Exp\ )
\end{aligned}
$$

$$
\begin{aligned}
Stat &\rightarrow \textbf{id} := Exp \\
Stat &\rightarrow Stat\ ;\ Stat \\
Stat &\rightarrow \textbf{if}\ Exp\ \textbf{then}\ Stat\ \textbf{else}\ Stat \\
Stat &\rightarrow \textbf{if}\ Exp\ \textbf{then}\ Stat
\end{aligned}
$$

Gramáticas libres de contexto - Derivaciones

- Una derivación consiste en considerar las producciones como reglas de reescritura.
- Cada que tengamos un no terminal, podemos reemplazarlo por la parte derecha de alguna producción. Podemos hacer esto una y otra vez, hasta que sólo nos queden terminales.
- La secuencia resultante de terminales es una cadena perteneciente al lenguaje definido por la gramática.

Gramáticas libres de contexto - Ejemplo de derivaciones

$$
\begin{aligned}
T &\to R \\
T &\to \mathtt{a}T\mathtt{c} \\
R &\to \\
R &\to R\mathtt{b}R
\end{aligned}
$$

$$
\begin{aligned}
&\underline{T} \\
\Rightarrow\ & \mathtt{a}\underline{T}\mathtt{c} \\
\Rightarrow\ & \mathtt{aa}\underline{T}\mathtt{cc} \\
\Rightarrow\ & \mathtt{aa}\underline{R}\mathtt{cc} \\
\Rightarrow\ & \mathtt{aa}R\mathtt{b}\underline{R}\mathtt{cc} \\
\Rightarrow\ & \mathtt{aa}\underline{R}\mathtt{bcc} \\
\Rightarrow\ & \mathtt{aa}R\mathtt{b}\underline{R}\mathtt{bcc} \\
\Rightarrow\ & \mathtt{aa}R\mathtt{b}\underline{R}\mathtt{b}R\mathtt{bcc} \\
\Rightarrow\ & \mathtt{aa}\underline{R}\mathtt{bb}R\mathtt{bcc} \\
\Rightarrow\ & \mathtt{aabb}\underline{R}\mathtt{bcc} \\
\Rightarrow\ & \mathtt{aabbbcc}
\end{aligned}
$$

$$
\begin{aligned}
&\underline{T} \\
\Rightarrow\ & \mathtt{a}\underline{T}\mathtt{c} \\
\Rightarrow\ & \mathtt{aa}\underline{T}\mathtt{cc} \\
\Rightarrow\ & \mathtt{aa}\underline{R}\mathtt{cc} \\
\Rightarrow\ & \mathtt{aa}\underline{R}\mathtt{b}R\mathtt{cc} \\
\Rightarrow\ & \mathtt{aa}\underline{R}\mathtt{b}R\mathtt{b}R\mathtt{cc} \\
\Rightarrow\ & \mathtt{aab}\underline{R}\mathtt{b}R\mathtt{cc} \\
\Rightarrow\ & \mathtt{aab}\underline{R}\mathtt{b}R\mathtt{b}R\mathtt{cc} \\
\Rightarrow\ & \mathtt{aabb}\underline{R}\mathtt{b}R\mathtt{cc} \\
\Rightarrow\ & \mathtt{aabbb}\underline{R}\mathtt{cc} \\
\Rightarrow\ & \mathtt{aabbbcc}
\end{aligned}
$$

Arboles de sintaxis

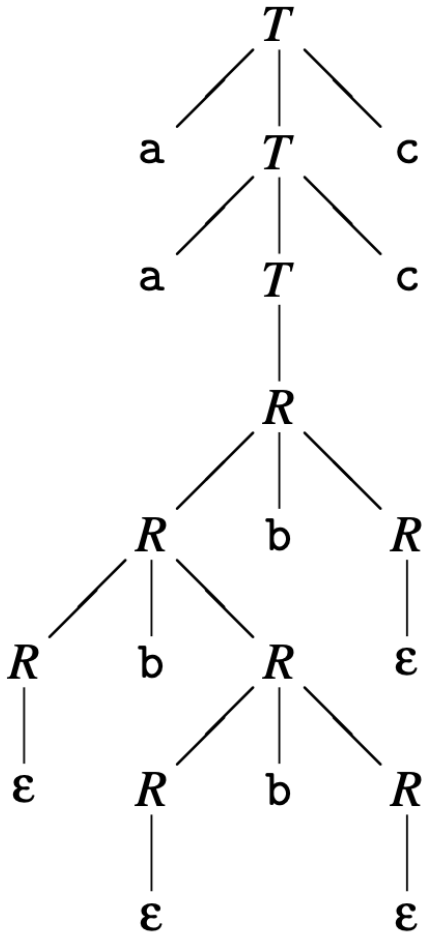
- Una derivación puede ser dibujada en forma de árbol. La raíz de este árbol es el símbolo inicial de la gramática y cada vez que reescribimos un terminal agregamos como sus hijos los símbolos en la parte derecha de la producción utilizada.

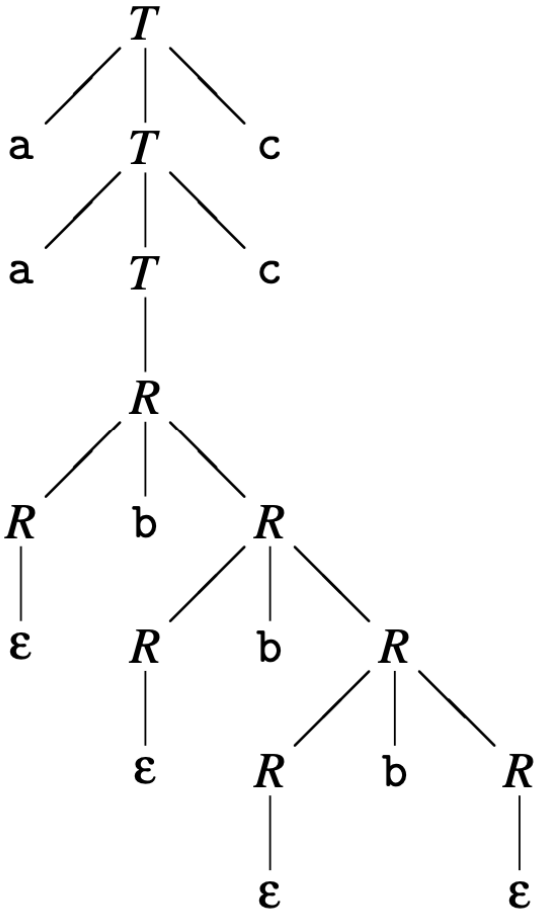
- Las hojas del árbol son terminales que al leerse de izquierda a derecha forman la cadena derivada.

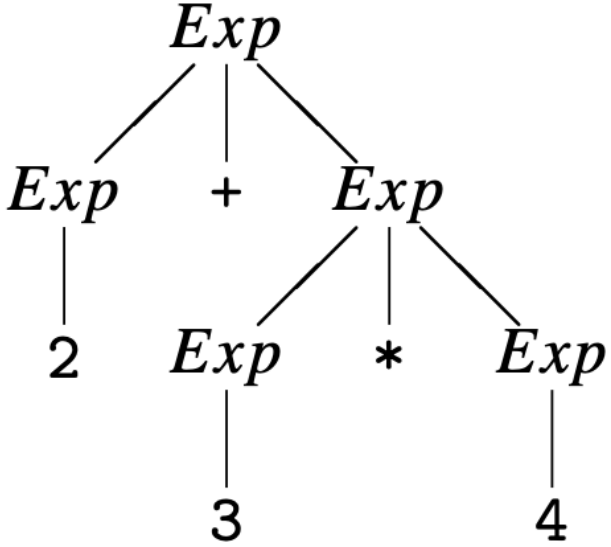
- Si un no terminal es reescrito usando una producción vacía, se muestra **ε** como su hijo.

- Este árbol se conoce por el nombre de **árbol de sintaxis**.

Arboles de sintaxis - Ejemplos







Arboles de sintaxis

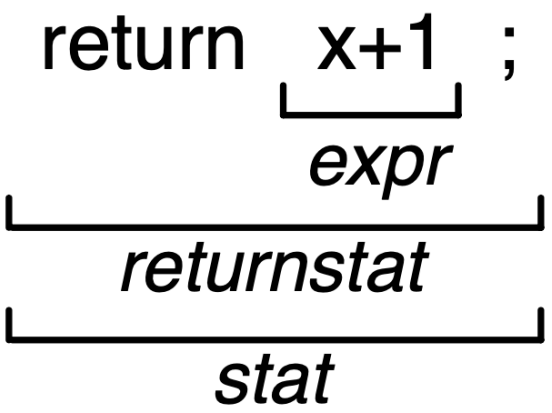
- La fase de análisis sintáctico de un compilador tomará una cadena de tokens producidos por el lexer, y a partir de estos construirá un árbol de sintaxis para la cadena de tokens encontrando una derivación a partir del símbolo inicial de la gramática.

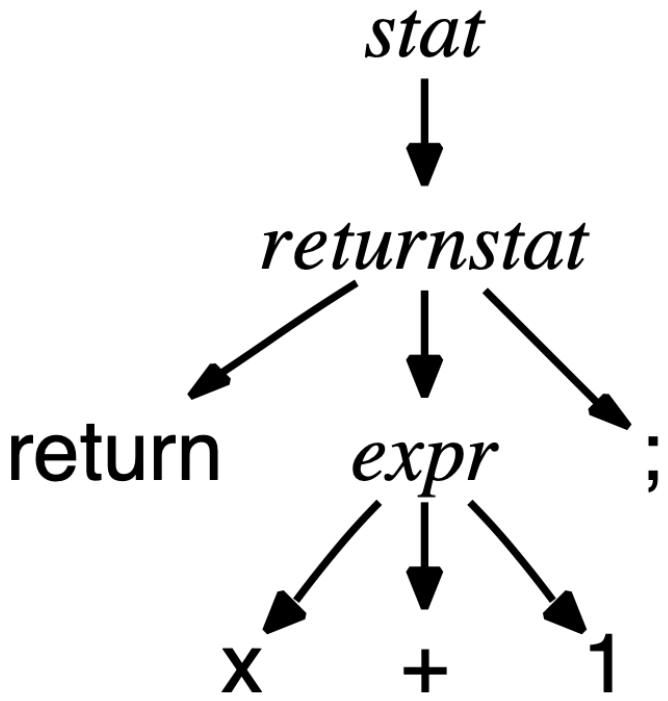
- Esto puede hacerse adivinando las derivaciones hasta encontrar la correcta, pero no es un método muy efectivo.

- Otros parsers lo hacen de manera contraria: Buscan partes de la cadena de entrada que empaten con la parte derecha de alguna producción. El árbol de sintaxis se completa de manera inversa.

Arboles de sintaxis - Identificación de frases

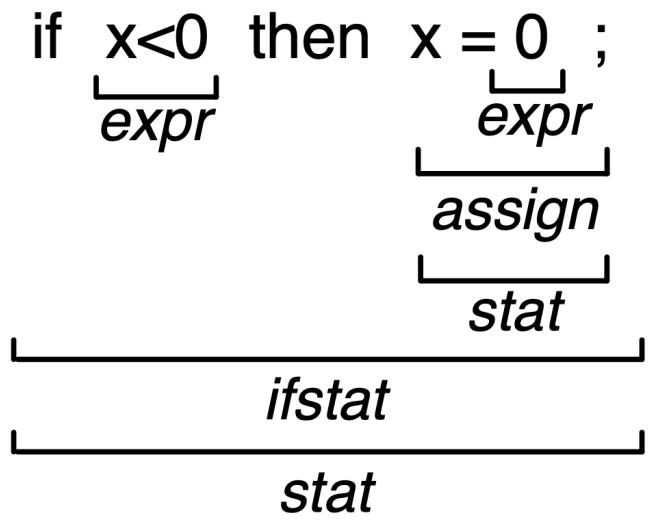
- Tomemos el ejemplo de la frase:
    - return x+1;
- La secuencia x+1 juega el papel de una expresión y la frase completa es un enunciado return.

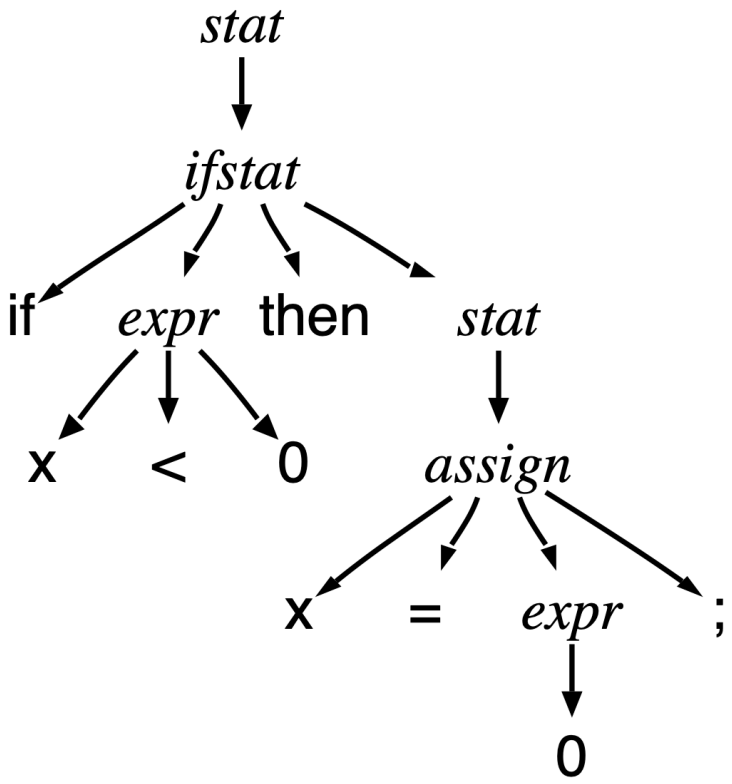




Arboles de sintaxis - Identificación de frases

- Tomemos otro ejemplo más complicado:





- Los árboles de sintaxis son importantes porque nos dicen todo lo que necesitamos saber acerca de la sintaxis de una frase.

Parser recursivo descendente

- Un parser verifica que un enunciado se apega a la sintaxis de un lenguaje.
- Para verificar la membresía al lenguaje, un parser identifica el árbol de sintaxis de un enunciado.
- Lo bueno es que el parser no construye realmente el árbol de sintaxis en la memoria. Es suficiente con reconocer las subestructuras y los tokens asociados.
- La mayoría del tiempo, sólo se necesita ejecutar algún código sobre los tokens en una subestructura.
- En la práctica, queremos que el parser “haga esto cuando vea aquello”.

Parser recursivo descendente

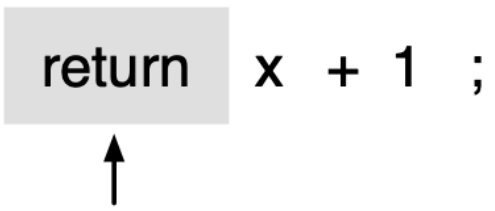
- Para evitar construir árboles de sintaxis, los rastreamos de manera implícita a través de una secuencia de invocaciones a funciones.
- Lo que hacemos es escribir una función para cada subestructura (nodo interior) del árbol de sintaxis.
- Cada función, digamos f, ejecuta código para empatar (match) con sus hijos.
- Para empatar una subestructura, f invoca a la función asociada con ese subarbol.
- Para empatar los tokens hijos, f invoca a la función de apoyo match().

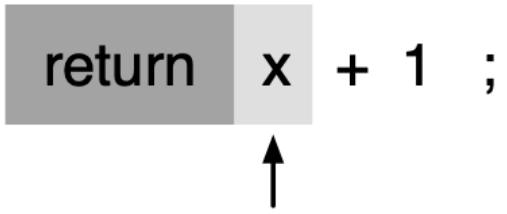
Parser recursivo descendente

- Siguiendo esta simple fórmula, llegamos a las siguientes funciones a partir del árbol de sintaxis para el enunciado return x + 1;

```
/** To parse a statement, call stat(); */
void stat()       { returnstat(); }
void returnstat() { match("return"); expr(); match(";"); }
void expr()       { match("x"); match("+"); match("1"); }
```

- La función match() avanza un cursor de entrada después de comparar el token de entrada actual contra su argumento.





Parser recursivo descendente

- Para ampliar el ejemplo, veamos como hacer el análisis sintáctico (parsing) de tres tipos de enunciados.

```
void stat() {
    if ( «lookahead token is return» )          returnstat();
    else if ( «lookahead token is identifier» ) assign();
    else if ( «lookahead token is if» )         ifstat();
    else «parse error»
}
```

- Este tipo de parser se conoce como “arriba-abajo” (top-down) porque inicia en la parte de arriba del árbol de sintaxis y se mueve para abajo hacia los nodos hoja.

Patrones - Mapeo de gramáticas a reconocedores recursos descendentes

Propósito.

- Este patrón describe como traducir una gramática hacia reconocedores recursivos descendentes que empatan enunciados en el lenguaje especificado.

- En este patrón se identifica el flujo de control básico para cualquier lexer, parser o árbol de sintaxis recursivo descendente.

Patrones - Mapeo de gramáticas a reconocedores recursos descendentes

Discusión.

- Al construir lexers y parsers a mano, el mejor punto de partida es una gramática. Las gramáticas son una manera concisa de expresar los lenguajes que se desean reconocer. Además, las gramáticas sirven para documentar en un manual de referencia o como comentarios en el código de un parser.

- Este patrón nos permite construir reconocedores directamente a partir de las gramáticas.

- Se debe considerar que este patrón funciona con muchas gramáticas, pero no con todas. Por ejemplo cuando hay reglas con recursión a la izquierda.

Patrones - Mapeo de gramáticas a reconocedores recursos descendentes

Implementación.

- Una gramática, G, es un conjunto de reglas a partir de las cuales generamos la definición de una clase con métodos para cada regla:

```
public class G extends Parser { // parser definition written in Java
    «token-type-definitions»
    «suitable-constructor»
    «rule-methods»
}
```

Patrones - Mapeo de gramáticas a reconocedores recursos descendentes

- La clase Parser define el estado típico que necesita un parser, como el siguiente token (lookahead token) y el flujo de entrada.
- Para cada regla, r, definida en una gramática, se construye un método con el mismo nombre:

```
public void r() {
    ...
}
```

Patrones - Mapeo de gramáticas a reconocedores recursos descendentes

- Las referencias a tokens para el token tipo T se convierten en invocaciones al método match(T). El método match() está incluido en la clase Parser y su función es consumir un token si el tipo del token actual es T y arrojar una excepción en caso contrario.

- También es necesario definir el tipo T en algún lado, ya sea en el parser o en el lexer. Para cada token T, se debe escribir:

```
public static final int T = «sequential-integer»;
```

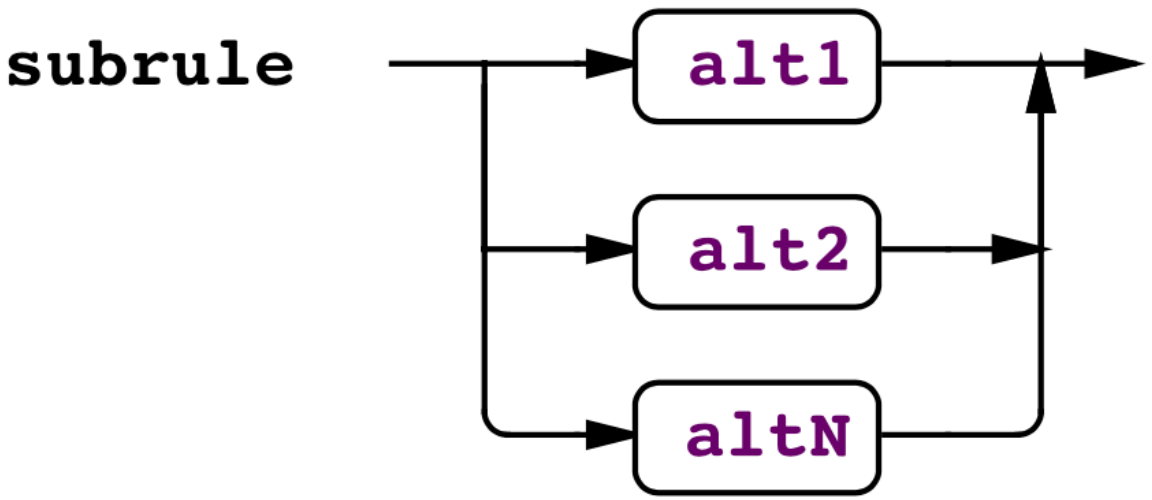
- Puede ser necesario también definir:

```
public static final int INVALID_TOKEN_TYPE = 0; // to be explicit
public static final int EOF = -1;               // EOF token type
```

Patrones - Mapeo de gramáticas a reconocedores recursos descendentes

- Las alternativas se convierten en secuencias switch o if-then-else, dependiendo de la complejidad de la decisión hacia adelante.
- Cada alternativa toma una expresión que predice si la alternativa será exitosa sobre la entrada actual. Consideremos la siguiente subregla genérica y su flujo de control:

```
(«alt1»|«alt2»|..|«altN»)
```



Patrones - Mapeo de gramáticas a reconocedores recursos descendentes

- Las implementación más general de la subregla es:

```
if ( «lookahead-predicts-alt1» ) { «match-alt1» }
else if ( «lookahead-predicts-alt2» ) { «match-alt2» }
...
else if ( «lookahead-predicts-altN» ) { «match-altN» }
else «throw-exception» // parse error (no viable alternative)
```

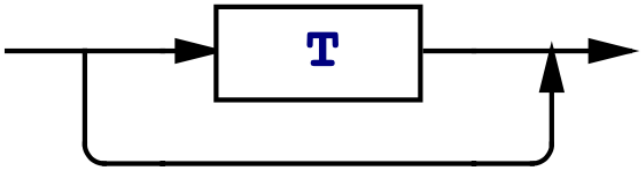
Patrones - Mapeo de gramáticas a reconocedores recursos descendentes

- Si sólo se analiza un símbolo hacia adelante, se puede utilizar un enunciado switch, el cual es normalmente más eficiente:

```
switch ( «lookahead-token» ) {
    case «token1-predicting-alt1» :
    case «token2-predicting-alt1» :
    ...
        «match-alt1»
        break;
    case «token1-predicting-alt2» :
    case «token2-predicting-alt2» :
    ...
        «match-alt2»
        break;
    ...
    case «token1-predicting-altN» :
    case «token2-predicting-altN» :
    ...
        «match-altN»
        break;
    default : «throw-exception»
}
```

Patrones - Mapeo de gramáticas a reconocedores recursos descendentes

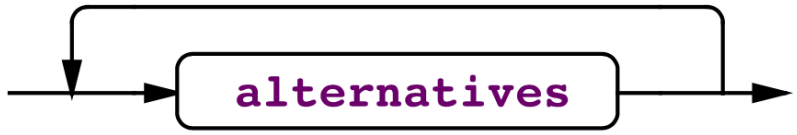
- Si la subregla es opcional:




```
if ( «lookahead-is-T» ) { match(T); } // no error else clause
```

Patrones - Mapeo de gramáticas a reconocedores recursos descendentes

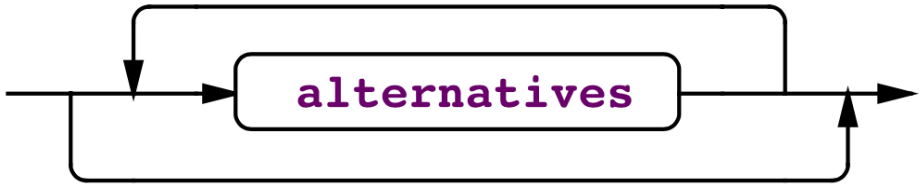
- El flujo de control para subreglas de uno o más:




```
do {
    «code-matching-alternatives»
} while ( «lookahead-predicts-an-alt-of-subrule» );
```

Patrones - Mapeo de gramáticas a reconocedores recursos descendentes

- El flujo de control para subreglas de uno o más:




```
while ( «lookahead-predicts-an-alt-of-subrule» ) {
    «code-matching-alternatives»
}
```

Patrones - Un parser recursivo descendente LL(1)

Propósito.

- Este patrón analiza la estructura de la secuencia de tokens de una frase visualizando el siguiente token de la secuencia.
- El parser pertenece a la clase de parsers LL(1) de arriba hacia abajo (top-down) porque visualiza sólo un token adelante.
- Este patrón es la base para otros patrones más complejos.

Patrones - Un parser recursivo descendente LL(1)

Discusión.

- El patrón muestra como implementar decisiones de parsing que visualizan sólo un token hacia adelante. Es la manera más débil, pero la más fácil de entender e implementar.

- Para tomar las decisiones de parsing, el parser prueba el token actual con los conjuntos de alternativas.

- El parser debe intentar la alternativa que puede iniciar con el token actual.

Patrones - Un parser recursivo descendente LL(1)

- Como ejemplo iniciemos con la forma más fácil de computar una vista hacia adelante (lookahead).

```
stat: 'if' ...     // lookahead set is {if}
    | 'while' ... // lookahead set is {while}
    | 'for' ...    // lookahead set is {for}
    ;

body_element
    : stat       // lookahead is {if, while, for}
    | LABEL ':' // lookahead is {LABEL}
    ;
```

Patrones - Un parser recursivo descendente LL(1)

```
optional_init
    : '=' expr
    | // empty alternative
    ;


decl: 'int' ID optional_init ';' ;
arg : 'int' ID optional_init ;
func_call: ID '(' arg ')' ; // calls arg; ')' included in lookahead
```

Patrones - Un parser recursivo descendente LL(1)

- Las decisiones de parsing pueden ser deterministas o indeterministas.

```
/** Match -3, 4, -2.1 or x, salary, username, and so on */
expr: '-'? (INT|FLOAT) // '-', INT, or FLOAT predicts this alternative
    | ID               // ID predicts this alternative
    ;
```

```
expr: ID '++'    // match "x++"
    | ID '--'    // match "x--"
    ;
```

No determinista

```
expr: ID ('++'|'--') ;   // match "x++" or "x--"
```

Determinista

Patrones - Un parser recursivo descendente LL(1)

- Se implementará un parser, ListParser, que se relaciona con el lexer implementado en el patrón anterior (un lexer recursivo descendente).

- La gramática es la siguiente:

```
list     : '[' elements ']' ;          // match bracketed list
elements : element (',' element)* ;    // match comma-separated list
element  : NAME | list ;               // element is name or nested l
```

Patrones - Un parser recursivo descendente LL(1)

- Siguiendo la gramática podemos codificar el siguiente parser.
- Las producciones (reglas) elements y element usan lookahead para tomar las decisiones de parsing.

```
public class ListParser extends Parser {
    public ListParser(Lexer input) { super(input); }

    /** list : '[' elements ']' ; // match bracketed list */
    public void list() {
        match(ListLexer.LBRACK); elements(); match(ListLexer.RBRACK);
    }
    /** elements : element (',' element)* ; */
    void elements() {
        element();
        while ( lookahead.type==ListLexer.COMMA ) {
            match(ListLexer.COMMA); element();
        }
    }
    /** element : name | list ; // element is name or nested list */
    void element() {
        if ( lookahead.type==ListLexer.NAME ) match(ListLexer.NAME);
        else if ( lookahead.type==ListLexer.LBRACK ) list();
        else throw new Error("expecting name or list; found "+lookahead);
    }
}
```

Patrones - Un parser recursivo descendente LL(1)

- Para apoyar esta clase concreta, debemos construir una clase abstracta Parser.

- Primero, necesitamos dos variables de estado: un flujo de entrada de tokens y el token actual del flujo.

```
Lexer input;      // from where do we get tokens?
Token lookahead; // the current lookahead token
```

- De manera alternativa, podríamos usar una estructura de datos que contuviera todos los tokens. De esta manera, podríamos rastrear un índice dentro de la estructura de datos.

Patrones - Un parser recursivo descendente LL(1)

- También, necesitamos métodos para comparar los token que se esperan contra el token actual en el flujo de tokens y consumir la entrada.

```
/** If lookahead token type matches x, consume & return else error */
public void match(int x) {
    if ( lookahead.type == x ) consume();
    else throw new Error("expecting "+input.getTokenName(x)+
                         "; found "+ lookahead);
}
public void consume() { lookahead = input.nextToken(); }
```

Patrones - Un parser recursivo descendente LL(1)

- Para probar el parser, debemos hacer una codificación como se muestra.
- Para cualquier entrada válida, el parser no regresa nada, pero se puede agregar código para rastrear la lista de nombres.

```
ListLexer lexer = new ListLexer(args[0]); // parse command-line arg
ListParser parser = new ListParser(lexer);
parser.list(); // begin parsing at rule list
```

- Si existiera un error, desearíamos arrojar una excepción como se muestra.

```
$ java Test '[a, ]'
Exception in thread "main" java.lang.Error:
  expecting name or list; found <']',RBRACK>
        at ListParser.element(ListParser.java:24)
        at ListParser.elements(ListParser.java:16)
        at ListParser.list(ListParser.java:8)
        at Test.main(Test.java:6)
$
```

Patrones - Un parser recursivo descendente LL(k)

Propósito.

- Este patrón analiza la estructura sintáctica de la secuencia de tokens de una frase usando k>1 tokens hacia adelante.

- Un parser LL(k) aumenta el patrón previo con hasta un número fijo k de tokens hacia adelante.

Patrones - Un parser recursivo descendente LL(k)

Discusión.

- La fuerza de un parser recursivo descendente depende enteramente de la fuerza de sus decisiones hacia adelante.
- Un solo token hacia adelante es muy débil en el sentido que usualmente tiene que adaptarse la gramática para LL(1).
- Al permitir un buffer más grande (pero fijo) para los tokens hacia adelante, obtenemos un parser suficientemente fuerte para la mayoría de los lenguajes de computadora. Esto incluye, archivos de configuración, formatos de datos, protocolos de red, lenguajes gráficos, entre otros.
- Algunos lenguajes de programación presentan retos más grandes para lo cual requerimos un patrón de reconocimiento más poderoso.

Patrones - Un parser recursivo descendente LL(k)

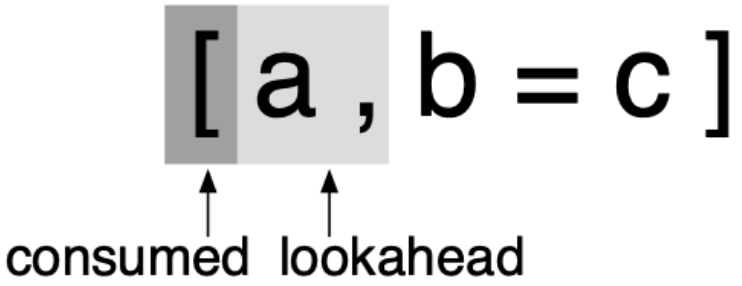
- Para entender porque necesitamos ver hacia adelante k>1 tokens, aumentemos la gramática del lenguaje de listas, permitiendo asignaciones como elementos de la lista. Por ejemplo, Queremos reconocer la lista [a, b=c, [d,e]]. Para acomodar este cambio, podemos agregar una alternativa a elemento que empate con asignación:

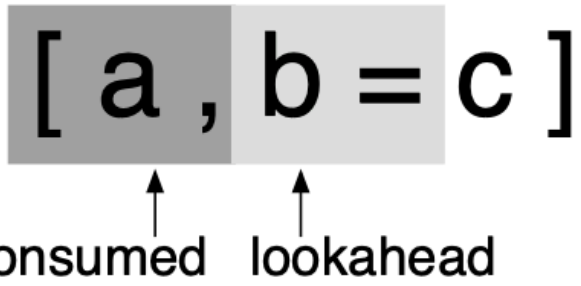
```
list     : '[' elements ']' ;          // match bracketed list
elements : element (',' element)* ;    // match comma-separated list
element  : NAME '=' NAME               // match assignment such as a=b
         | NAME
         | list
         ;
```

Patrones - Un parser recursivo descendente LL(k)

- Esa nueva alternativa hace que **element** sea no LL(1) dado que las dos primeras alternativas empiezan con el mismo token NAME. Ahora necesitamos dos tokens hacia adelante para distinguir las alternativas. La gramática es LL(2). Cada que buscamos un **element**, es necesario decidir si es una asignación o solo un nombre. Si la secuencia hacia adelante es un token NAME seguido de =, la decisión del parser debería predecir la primera alternativa. De otro modo, el parser debería predecir la segunda alternativa.

- Por ejemplo, el siguiente diagrama representa el token hacia adelante disponible para la decisión de parser en **element** para la entrada [a, b=c]:





Patrones - Un parser recursivo descendente LL(k)

- Si solo tuviéramos un parser LL(1), tenemos que reescribir element para que luzca así:

```
element  : NAME ('=' NAME)?  // match assignment such as a=b or just a
         | list
         ;
```

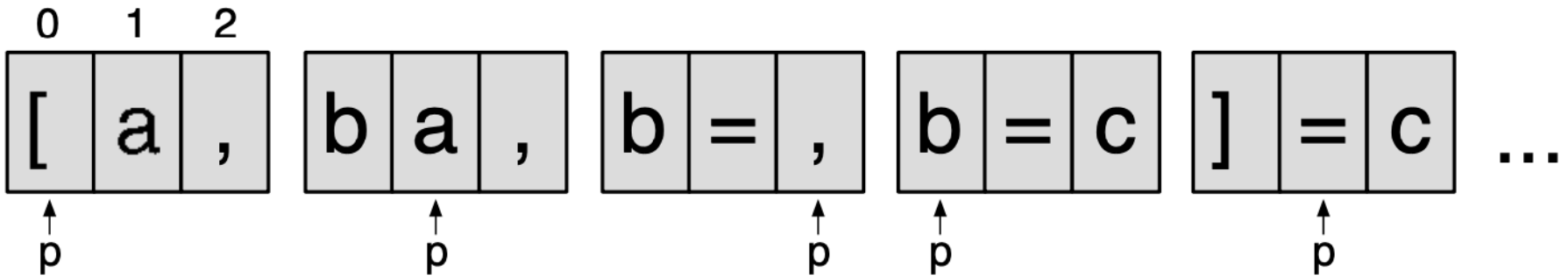
- Esta subregla opcional (..)? funciona, pero es menos clara. Existen muchas situaciones similares que ocurren en gramáticas reales.

Patrones - Un parser recursivo descendente LL(k)

- La manera más simple de proveer varios tokens hacia adelante es crear un buffer para toda la entrada de tokens. El cursor de entrada puede ser un índice entero dentro del buffer, digamos, p. Podemos ejecutar p++ para consumir los tokens. Los siguientes k tokens serán tokens[p]..tokens[p+k-1].

- Este enfoque funciona para una entrada razonablemente pequeña.

- Gráficamente, aquí se muestra como se mueve p sobre el buffer según el parser va consumiendo tokens de la entrada [a, b=c]:



Patrones - Un parser recursivo descendente LL(k)

- Para implementar el lenguaje de listas aumentado con asignaciones, primero construiremos la infraestructura para ver hacia adelante, y después implementaremos la regla LL(2) para element.

- Para apoyar un buffer fijo, la clase Parser necesita los siguientes campos:

```
Lexer input;        // from where do we get tokens?
Token[] lookahead; // circular lookahead buffer
int k;              // how many lookahead symbols
int p = 0;          // circular index of next token position to fill
```

Patrones - Un parser recursivo descendente LL(k)

- Debido a que diferentes parsers necesitan diferentes cantidades de tokens hacia adelante, el constructor toma un argumento **size** e inicia el buffer.

```
public Parser(Lexer input, int k) {
    this.input = input;
    this.k = k;
    lookahead = new Token[k];           // make lookahead buffer
    for (int i=1; i<=k; i++) consume(); // prime buffer with k lookahead
}
```

- Para consumir un token, el parser avanza el índice de forma circular y agrega otro token al buffer.

```
public void consume() {
    lookahead[p] = input.nextToken();   // fill next position with token
    p = (p+1) % k;                      // increment circular index
}
```

Patrones - Un parser recursivo descendente LL(k)

- Para aislar el mecanismo hacia adelante dentro del parser, es una buena idea crear métodos especiales: LA( ) y LT( ). El método LA( ) regresa hasta k tokens hacia adelante iniciando en k=1.
- Recordemos que los tokens son enteros que representan las categorías de los símbolos en la entrada.
- El método LT() regresa el token hacia adelante en una profundidad particular.

```
public Token LT(int i) {return lookahead[(p+i-1) % k];} // circular fetch
public int LA(int i) { return LT(i).type; }
public void match(int x) {
    if ( LA(1) == x ) consume();
    else throw new Error("expecting "+input.getTokenName(x)+
                         "; found "+LT(1));
}
```

Patrones - Un parser recursivo descendente LL(k)

- La regla única para element necesita más de un simple token hacia adelante en nuestro caso.
- La implementación del método prueba los dos primeros tokens hacia adelante, LA(1) y LA(2), para predecir la primera alternativa:

```
/** element : NAME '=' NAME | NAME | list ; assignment, NAME or list */
void element() {
    if ( LA(1)==LookaheadLexer.NAME && LA(2)==LookaheadLexer.EQUALS ) {
        match(LookaheadLexer.NAME);
        match(LookaheadLexer.EQUALS);
        match(LookaheadLexer.NAME);
    }
    else if ( LA(1)==LookaheadLexer.NAME ) match(LookaheadLexer.NAME);
    else if ( LA(1)==LookaheadLexer.LBRACK ) list();
    else throw new Error("expecting name or list; found "+LT(1));
}
```

Patrones - Un parser recursivo descendente LL(k)

- Para probar la implementación del parser LL(2), invocamos al lexer y al parser correspondientes.

```
LookaheadLexer lexer = new LookaheadLexer(args[0]); // parse arg
LookaheadParser parser = new LookaheadParser(lexer, 2);
parser.list(); // begin parsing at rule list
```

Patrones - Un parser recursivo descendente

```
public class LookaheadLexer extends Lexer {
    public static int NAME = 2;
    public static int COMMA = 3;
    public static int LBRACK = 4;
    public static int RBRACK = 5;
    public static int EQUALS = 6;
    public static String[] tokenNames = {"n/a", "<EOF>", "NAME", "COMMA", "LBRACK", "RBRACK", "EQUALS"};

    public String getTokenName(int x) {
        return tokenNames[x];
    }

    public LookaheadLexer(String input) {
        super(input);
    }

    boolean isLETTER() {
        return c>='a' && c<='z' || c>='A' && c<='Z';
    }

    public Token nextToken() {
        while(c!=EOF){
            switch(c){
                case ' ': case '\t': case '\n': case '\r': WS(); continue;
                case ',': consume(); return new Token(COMMA,",");
                case '[': consume(); return new Token(LBRACK,"[");
                case ']': consume(); return new Token(RBRACK,"]");
                case '=': consume(); return new Token(EQUALS,"=");
                default:
                    if(isLETTER())
                        return NAME();

                    throw new Error("Invalid character: " + c);
            }
        }

        return new Token(EOF_TYPE,"<EOF>");
    }
```

Patrones - Un parser recursivo descendente

```
    Token NAME() {
        StringBuilder buf = new StringBuilder();

        do {
            buf.append(c);
            consume();
        } while (isLETTER());

        return new Token(NAME,buf.toString());
    }

    void WS() {
        while (c==' ' || c == '\t' || c == '\n' || c == '\r')
            consume();
    }
}
```

Patrones - Un parser recursivo descendente

```
public class Parser2 {
    Lexer input;
    Token[] lookahead;
    int k;
    int p = 0;

    public Parser2(Lexer input, int k) {
        this.input = input;
        this.k = k;
        lookahead = new Token[k];

        for (int i=1; i<=k; i++)
            consume();
    }

    public void consume() {
        lookahead[p] = input.nextToken(); // fill next position with token
        p = (p+1) % k;
    }

    public Token LT(int i) {
        return lookahead[(p+i-1) % k];
    } // circular fetch

    public int LA(int i) {
        return LT(i).type;
    }

    public void match(int x) {
        if ( LA(1) == x )
            consume();
        else
            throw new Error("Expecting " + input.getTokenName(x) + "; found " + LT(1));
    }
}
```

```
public class LookaheadParser extends Parser2 {
    public LookaheadParser(Lexer input, int k) {
        super(input, k);
    }

    public void list() {
        match(LookaheadLexer.LBRACK);
        elements();
        match(LookaheadLexer.RBRACK);
    }

    void elements() {
        element();

        while (LA(1) == LookaheadLexer.COMMA) {
            match(LookaheadLexer.COMMA);
            element();
        }
    }

    void element() {
        if (LA(1) == LookaheadLexer.NAME && LA(2) == LookaheadLexer.EQUALS) {
            match(LookaheadLexer.NAME);
            match(LookaheadLexer.EQUALS);
            match(LookaheadLexer.NAME);
        }
        else if (LA(1)==LookaheadLexer.NAME )
                match(LookaheadLexer.NAME);
        else if (LA(1)==LookaheadLexer.LBRACK)
            list();
        else throw new Error("Expecting name or list; found " + LT(1));
    }
}
```

Patrones - Un parser recursivo descendente

Patrones - Un parser recursivo descendente LL(k)

- Para probar la implementación del parser LL(2), invocamos al lexer y al parser correspondientes.

```
$ java Test '[a,b=c,[d,e]]'
$
```

```
$ java Test '[a,b=c,,[d,e]]'
Exception in thread "main" java.lang.Error:
 expecting name or list; found <',',,>
        at LookaheadParser.element(LookaheadParser.java:25)
        at LookaheadParser.elements(LookaheadParser.java:12)
        at LookaheadParser.list(LookaheadParser.java:7)
        at Test.main(Test.java:6)
$
```